Informatyka dla Matematyków

Włodzimierz Moczurad

18 stycznia 2024

Spis treści

1 Wstęp **3**
- 1.1 Algorytmy i maszyny Turinga . . . 3
  - 1.1.1 Algorytmika . . . 3
  - 1.1.2 Od Hilberta do Turinga . . . 5
  - 1.1.3 Maszyna Turinga . . . 7

2 Teoria języków i automatów **11**
- 2.1 Podstawowe pojęcia . . . 11
  - 2.1.1 Przykłady słów, prefiksy, sufiksy . . . 11
  - 2.1.2 Przykłady języków formalnych . . . 12
  - 2.1.3 Grafy . . . 12
  - 2.1.4 Drzewa . . . 13
- 2.2 Języki regularne . . . 14
  - 2.2.1 Deterministyczny automat skończony (DFA) . . . 14
  - 2.2.2 Operacje regularne . . . 17
  - 2.2.3 Niedeterministyczny automat skończony (NFA) . . . 18
  - 2.2.4 Zamkniętość ze względu na operacje regularne . . . 20
  - 2.2.5 Wyrażenia regularne . . . 22
- 2.3 Języki nieregularne . . . 23
  - 2.3.1 Gramatyki bezkontekstowe i kontekstowe . . . 24
- 2.4 Problemy decyzyjne i hierarchia Chomsky'ego . . . 26
  - 2.4.1 Przykłady problemów decyzyjnych . . . 26
  - 2.4.2 Typy gramatyk — porównanie . . . 26
  - 2.4.3 Problemy decyzyjne — porównanie . . . 27
  - 2.4.4 Hierarchia Chomsky'ego . . . 31

3 Złożoność obliczeniowa **32**
- 3.1 Złożoność — zużywane zasoby . . . 32
- 3.2 Notacja $\mathcal{O}$ i o . . . 33
- 3.3 Modele obliczeń . . . 34
- 3.4 Klasy złożoności . . . 35
- 3.5 Hierarchia klas złożoności . . . 37
- 3.6 Koszt algorytmu a złożoność problemu . . . 37
  - 3.6.1 Proste algorytmy sortowania . . . 38
  - 3.6.2 Dolne ograniczenie złożoności czasowej sortowania . . . 41

4 Reprezentacja maszynowa liczb **43**
- 4.1 Liczby całkowite . . . 43
  - 4.1.1 Zagadki C-owe . . . 43
  - 4.1.2 Kodowanie liczb całkowitych . . . 44
  - 4.1.3 Zakresy wartości . . . 46
  - 4.1.4 Liczby całkowite w języku C — parę uwag . . . 46

4.1.5 Wartości ze znakiem i bez znaku . . . 47
4.1.6 Konwersje liczb . . . 47
4.1.7 Powielanie bitu znaku . . . 50
4.1.8 Kiedy używać liczb bez znaku? . . . 52
4.1.9 Zmiana znaku przez negację i inkrementację . . . 52
4.1.10 Operacje na liczbach całkowitych . . . 53
4.1.11 Własności arytmetyki komputerowej . . . 59
4.2 Reprezentacja zmiennopozycyjna . . . 60
4.2.1 Zagadki C-owe . . . 60
4.2.2 Reprezentacja dwójkowa ułamków . . . 60
4.2.3 Definicja reprezentacji zmiennopozycyjnej . . . 61
4.2.4 Zakresy liczb i własności zp . . . 66
4.2.5 Operacje na liczbach zp . . . 67
4.2.6 Liczby zp w języku C . . . 70

5 Teoria informacji **71**
5.1 Miara informacji i entropia . . . 71
5.2 Kodowanie komunikatów i kompresja . . . 73
5.3 Średnia długość kodowania i redundancja . . . 74
5.4 Kodowanie optymalne — algorytm Huffmana . . . 75

Literatura **78**

Indeks **79**

Rozdział 1

Wstęp

Zasadniczym tematem skryptu jest wprowadzenie studentów matematyki w świat *informatyki*. Niniejszy skrypt stanowi swego rodzaju kompendium tych podstawowych wiadomości z dziedziny informatyki, które nie są jednocześnie częścią klasycznego kanonu studiów matematycznych, a które z wysokim prawdopodobieństwem w toku tych studiów okażą się przydatne. Zdecydowanie należy podkreślić, że prezentujemy tutaj jedynie zarys elementarnej wiedzy informatycznej, a nie szczegółowe omówienie tematów. Zagadnienia poruszane tutaj stanowią wstęp do odrębnych przedmiotów, takich jak *Algorytmy i struktury danych*, *Teoria języków i automatów*, *Obliczalność i złożoność*, *Teoria informacji*. Studenci informatyki większość z tych przedmiotów mają w kanonie przedmiotów obowiązkowych, ale studenci matematyki bądź matematyki komputerowej zapewne nie spotkają się z tą tematyką, o ile nie zechcą poszerzyć uzyskanej tutaj wiedzy w ramach przedmiotów fakultatywnych.

Skrypt podzielony jest na kilka rozdziałów, które w dużej mierze są niezależne. Poszczególne rozdziały można traktować jako wstęp i zachętę do wzięcia udziału w oddzielnych kursach dotyczących omawianej tematyki.

1.1 Algorytmy i maszyny Turinga

1.1.1 Algorytmika

Algorytmika jest niezwykle istotną, o ile nie najważniejszą częścią informatyki. Zajmuje się analizą i projektowaniem algorytmów, ale także tym, jakie struktury danych dobierać i jak analizować zachowanie się programów.

Tak jak wiele innych dziedzin nauki, algorytmika ma swoje źródła w starożytności: pojęcie algorytmu[1] jako ciągu instrukcji do wykonania pojawiło się już w czasach starożytnej Grecji. Jeden z najbardziej znanych, powstały około 300 lat p.n.e., to *algorytm Euklidesa,* służący do znajdowania największego wspólnego dzielnika danej pary liczb. Algorytm ten ma ogromne znaczenie, stosuje się go na przykład w kryptografii.

Okazuje się, że nie ma jednego wzoru na NWD(m, n), który działałby dla dowolnych m i n, ale Euklides zauważył, że jeżeli od większej liczby odejmiemy mniejszą, to ich NWD się nie zmieni. Bazując na tej obserwacji, stworzył algorytm, który można opisać tak:

```
wczytaj liczby m i n;
dopóki n > 0 wykonuj:
   jeżeli m > n, to zamień m i n;
```

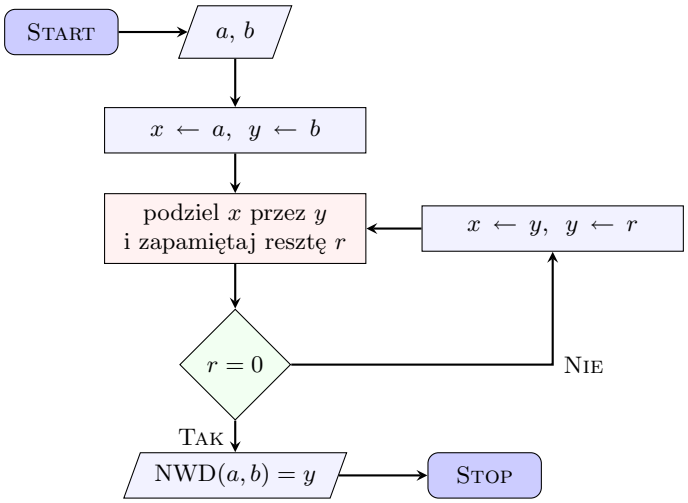
[1] Słowo *algorytm* pochodzi od nazwiska perskiego matematyka Mohammeda ibn Mûsâ al-Chuwârizmiego, urodzonego ok. 780 roku i zmarłego ok. 850 roku. Dziełem al-Chuwârizmiego, w którym algorytmy odgrywały największą rolę, jest paradoksalnie praca poświęcona algebrze *Al-Kitab al-mukhtasar fi hisab al-jabr wa-l-muqabala* (Krótka księga o obliczaniu przez dopełnienie i równoważenie).

`zastąp` n `różnicą` $n-m$`;`
`wypisz` m`.`

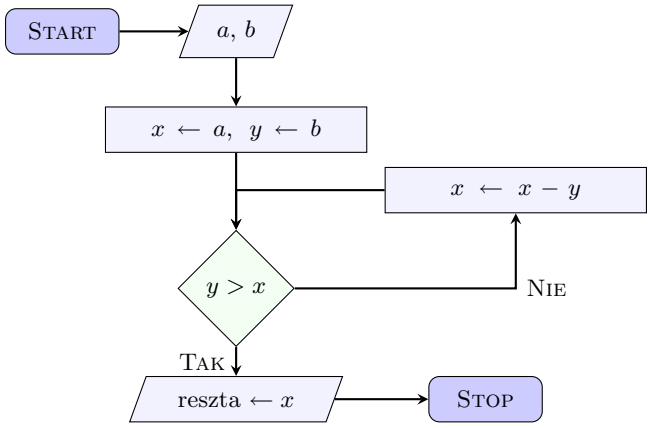
Napisanie algorytmu to dopiero pierwszy krok. W praktyce pojawia się pytanie, czy algorytm jest dobry, a przynajmniej akceptowalny. A co to znaczy dobry?...

Zastanówmy się, ile czasu zajęłoby komputerowi wykonanie powyższego algorytmu np. dla $m = 1$ i $n = 10^{30}$? Dla takich danych musimy wykonać 10^{30} odejmowań. Przyjmijmy optymistycznie, że procesor wykonuje ok. 10^{10} podstawowych operacji na sekundę,[2] czyli taka liczba odejmowań zajmie 10^{20} sekund (ponad 3×10^{20} lat). Wygląda dość przerażająco — np. w szyfrowaniu RSA stosuje się liczby rzędu 10^{200}! Jednak jeżeli zamiast odejmowania zastosujemy (nieco bardziej czasochłonną) operację mod (reszta z dzielenia), to całość obliczeń wyniesie poniżej jednej sekundy: wystarczy raz policzyć resztę. Na rysunku 1.1 przedstawiony jest (w postaci schematu blokowego) algorytm Euklidesa, z wykorzystaniem reszty z dzielenia.

*



Rysunek 1.1: Schemat blokowy algorytmu Euklidesa.



Rysunek 1.2: Podprocedura obliczania reszty z dzielenia.

Czy teraz jesteśmy usatysfakcjonowani nowym algorytmem? Niekoniecznie, gdyż nie wiemy, czy na pewno zastąpienie prostego odejmowania bardziej skomplikowaną operacją mod przynosi korzyść dla dowolnych danych. Być może istnieją dane, dla których nowy algorytm działa jeszcze gorzej niż poprzednio?

Liczbę operacji, które algorytm wykona w najgorszym możliwym przypadku (czyli dla najbardziej złośliwych danych), nazywamy *złożonością pesymistyczną algorytmu*.

[2] Obecnie (sierpień 2023) pojedynczy procesor jest w stanie wykonać kilka miliardów podstawowych operacji na sekundę.

Niewątpliwie, dla danych, w których jedna liczba jest jedynką, nowy algorytm działa niezwykle sprawnie (wykonuje tylko jedną operację mod), więc nie są to dane, przy których moglibyśmy ocenić złożoność pesymistyczną nowego algorytmu Euklidesa. Spróbujmy poszukać najgorszych danych, czyli takich, dla których algorytm wykona najwięcej pętli. Zauważmy, że:

- aby wykonać jeden obieg pętli, wystarczy para $(1, 1)$;
- aby wykonać dwa obiegi, potrzebujemy co najmniej $(3, 2)$, bo dzielnik musi być większy od 1, a dzielna od dzielnika;
- trzy obiegi: aby po pierwszym obiegu otrzymać „najtańszą" parę, która wymaga dwóch obiegów (czyli parę $(3, 2)$), dzielnikiem musi być 3, a najmniejsza dzielna dająca przy tym dzielniku resztę 2 i od niego większa to 5;
- stosując to samo rozumowanie otrzymujemy, że dla 4 obiegów potrzeba co najmniej $(8, 5)$;
- itd.

Liczby układają się w ciąg Fibonacciego:[3]

liczba obrotów	1	2	3	4	5	6	7	...
dzielnik:	1	2	3	5	8	13	21	...
dzielna:	1	3	5	8	13	21	34	...
nr liczby Fibonacciego:		F_4	F_5	F_6	F_7	F_8	F_9	...

Poprzednio rozważaliśmy 30-cyfrową liczbę 10^{30} i para $(1, 10^{30})$ — najgorsza dla algorytmu z odejmowaniem — potrzebowała 10^{30} obiegów pętli. Jak widać, dla algorytmu z resztą dzielenia, najgorsze są duże liczby Fibonacciego. Okazuje się, że liczby te rosną bardzo szybko i już $F_{146} > 10^{30}$, zatem nowy algorytm potrzebuje nie więcej niż 150 obiegów pętli, aby obsłużyć liczby rzędu 10^{30}. Nawet biorąc pod uwagę, że operacja mod jest zazwyczaj znacznie bardziej kosztowna niż odejmowanie, nowy algorytm jest istotnie lepszy.

Dobrze napisany algorytm ma znacznie większy wpływ na czas działania programu niż użyty sprzęt. Algorytmika zajmuje się poszukiwaniem i badaniem algorytmów. Ale zdarza się, że pomimo intensywnych wysiłków, nie udaje się rozwiązać problemu w satysfakcjonujący sposób. Wówczas przydają się pewne pojęcia z teorii złożoności lub obliczalności.

Podsumowując: algorytmy znane były już w starożytnej Grecji, swoją nazwę otrzymały ponad 1000 lat później i zawdzięczają ją średniowiecznemu matematykowi ze Wschodu, jednak nim zyskały formalną definicję, trzeba było poczekać kolejne 1000 lat. Definicja algorytmu jest już wkładem współczesnej nauki.

1.1.2 Od Hilberta do Turinga

Od zarania świata ludzkość boryka się z problemami, których w danym momencie nie potrafi rozwiązać. Na przykład, starożytni nie potrafili poradzić sobie z kwadraturą koła (wyznaczeniem boku kwadratu o polu równym polu koła o promieniu 1) — w XIX wieku pokazano, że metodami dopuszczanymi przez starożytnych nie da się tego zrobić.

Przełom wieków XIX i XX to czas powstawania nowych pojęć w matematyce, intensywny rozwój logiki i filozofii matematyki inspirowany między innymi próbami wyjaśnienia paradoksów, np.

- *paradoksu kłamcy*: pewien człowiek mówi „Ja teraz kłamię"; czy jest on kłamcą, czy nie?
- *paradoksu zbioru wszystkich zbiorów* (Cantor 1899);
- *paradoksu fryzjera* (Russell 1901): fryzjer, mieszkaniec pewnego miasta, goli tych jego mieszkańców, którzy sami się nie golą; czy fryzjer goli się sam?

Źródłem takich paradoksów było m.in. nieprecyzyjne stosowanie pojęć, a także brak rozróżnienia języka matematyki od języka metamatematyki, tzn. tego, w którym mówimy o tym pierwszym. Zakładano, że istnieje jeden zupełny język, w którym można wyrazić całą matematykę.

[3] $F_0 = 0$, $F_1 = 1$ i $F_n = F_{n-1} + F_{n-2}$ dla $n > 1$.

Rysunek 1.3: David Hilbert (1863–1943)[4]

Hilbert chciał w pełni zaksjomatyzować i sformalizować proces dowodzenia; pytał o istnienie algorytmu, któremu podamy zdanie zapisane w formalny sposób i który rozstrzygnie, czy to zdanie jest prawdziwe, czy fałszywe. Jest to tak zwany *Entscheidungsproblem* (dylemat decyzyjny, od niem. *entscheiden* — decydować). Hilbert był przekonany, że logika jest spójna i zupełna, tzn. każda dowiedlna formuła logiczna jest prawdziwa i każda prawdziwa formuła jest dowiedlna (istnieje dla niej formalny dowód, zgodny z przyjętymi regułami).

Rysunek 1.4: Kurt Gödel (1906–1978)[5]

Cios projektowi Hilberta zadał w 1930 roku młody austriacki logik Kurt Gödel, który udowodnił twierdzenie o niezupełności. Gödel pokazał, że (prawie) żaden system matematyczny (tzn. zbiór aksjomatów i reguł dowodzenia) nie jest zupełny. To „prawie" dotyczy wymagania, że system musi być wystarczająco silny (Gödel przeprowadził dowód twierdzenia o niezupełności, kodując zdania jako liczby, dlatego konieczne było wymaganie, aby w systemie logicznym można było zakodować arytmetykę) i być logicznie spójny. W systemie F możemy zapisać formułę G:

Zdanie G nie jest dowiedlne w F.

Zobaczmy, jak wygląda sprawa dowodu powyższego zdania:

- jeżeli powyższe zdanie ma dowód, to znaczy, że potrafimy udowodnić coś, co nie jest prawdziwe — sprzeczność;

[4]Wikipedia, zdjęcie w domenie publicznej
[5]Wikipedia, zdjęcie w domenie publicznej

- jeżeli powyższe zdanie nie ma dowodu, to znaczy, że istnieje zdanie prawdziwe, którego nie potrafimy udowodnić.

To wydarzenie miało znamienne skutki: pojawiła się potrzeba uzasadnienia, że dla pewnego problemu nie istnieje żaden algorytm. Dopóki sądzono, że odpowiedź na Entscheidungsproblem jest pozytywna, nie potrzebowano formalnej definicji algorytmu — wystarczyłoby go napisać. Ale udowodnienie, że żaden algorytm czegoś nie zrobi, wymagało sformalizowania pojęcia algorytmu. Zrobił to w 1936 roku Alan Turing, przedstawiając koncepcję *maszyny Turinga*.

Rysunek 1.5: Alan Turing (1912–1954)[6]

Zadał pytanie, jaka powinna być ogólna postać maszyny wykonującej algorytmy (obliczenia zdefiniowane w skończony sposób)? Powinna:

- mieć skończoną postać (skończoną liczbę stanów wewnętrznych),
- ale być zdolna prowadzić obliczenia na dowolnie dużych danych i wyprodukować dowolnie duży wynik.

Okazuje się, że pomysł prostej, abstrakcyjnej maszyny, która wykonuje program, przesuwając **głowicę** wzdłuż **taśmy** (czytając i zapisując pojedyncze symbole) jest wystarczający, aby spełnić te oczekiwania. Jest też wystarczająco silny, aby stanowić model ogólnej maszyny, na której można zrealizować dowolny algorytm. Współczesne komputery stanowią bardzo dobre przybliżenie idealnej maszyny Turinga: mają skończony software (oprogramowanie) i dający się rozbudowywać praktycznie w nieskończoność hardware (fizyczne elementy), ale są zbyt skomplikowane, aby tworzyć w oparciu o nie teorię matematyczną.

1.1.3 Maszyna Turinga

Wiemy już, że maszyna Turinga posiada taśmę, głowicę oraz program. Zakładamy, że taśma jest obustronnie nieskończona. W każdym kroku głowica przesuwa się w prawo, w lewo bądź stoi w miejscu, czyta i zapisuje jeden symbol zgodnie z programem.

Czym w takim razie jest program maszyny Turinga?

Definicja 1.1.1. *Deterministyczna maszyna Turinga* M to szóstka $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, gdzie:

Q — skończony zbiór *stanów* maszyny,

Σ — skończony zbiór *symboli wejściowych*,

Γ — skończony zbiór *symboli taśmowych*, przy czym $\Sigma \subseteq \Gamma$ oraz $Q \cap \Gamma = \varnothing$; dodatkowo wyróżniamy symbol pusty (blank, $\sqcup$), który nie należy do symboli wejściowych, *

[6] National Portrait Gallery, London, licencja `http://creativecommons.org/licenses/by-nc-nd/3.0/`

δ — *funkcja przejścia,* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, -\}$, (symbole L, R i $-$ opisują ewentualne przesunięcie głowicy: w lewo, w prawo lub bez przesunięcia),

q_0 — *stan początkowy,* $q_0 \in Q$,

F — zbiór *stanów końcowych* $F \subseteq Q$ (obliczenie maszyny kończy się, kiedy osiągnięty zostanie któryś ze stanów z F).

Przyjmujemy, że maszyna rozpoczyna działanie w stanie początkowym q_0; na taśmie jest słowo wejściowe, a poza tym cała taśma jest wypełniona blankami. Głowica znajduje się pod skrajnie lewym symbolem wejściowym. Taśma jest obustronnie nieskończona. Obliczenie maszyny kończy się, kiedy osiągnięty zostanie któryś ze stanów końcowych.

Wartość $\delta(A,\ x) = (B,\ y,\ m)$ oznacza, że maszyna będąca w stanie A, czytając symbol x z taśmy, przejdzie do stanu B, zapisze symbol y na taśmie i przesunie głowicę o jedną komórkę w lewo ($m = L$), w prawo ($m = R$) lub stoi ($m = -$).

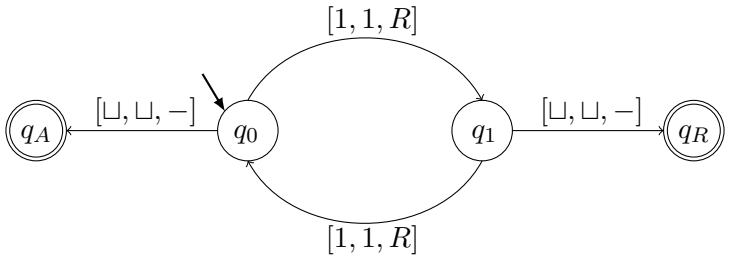
Przykład 1. Deterministyczna maszyna Turinga sprawdzająca, czy słowo wejściowe (złożone z jedynek) ma parzystą długość. Jeżeli tak, to maszyna kończy działanie w stanie końcowym q_A, jeżeli ma długość nieparzystą — w stanie q_R.

Opis maszyny:

- $Q = \{q_0, q_1, q_A, q_R\}$,
- $\Gamma = \{1, \sqcup\}$,
- stan początkowy q_0,
- $F = \{q_A, q_R\}$,
- funkcję przejścia δ można przedstawić w postaci tabelki:

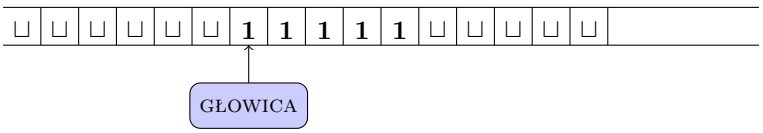
stan	$\sqcup$	1
q_0	$(\sqcup, -, q_A)$	$(1, R, q_1)$
q_1	$(\sqcup, -, q_R)$	$(1, R, q_0)$

lub w postaci grafu:



Rysunek 1.6: Graf funkcji przejścia maszyny Turinga sprawdzającej parzystą długość słowa wejściowego; strzałka wskazuje stan początkowy, a podwójny okrąg oznacza stan końcowy

Jak zadziała maszyna na pewnym słowie wejściowym? Maszyna rozpoczyna działanie w stanie q_0, mając głowicę ustawioną na pierwszym symbolu słowa.



Rysunek 1.7: Taśma maszyny Turinga z głowicą ustawioną na pierwszym symbolu słowa wejściowego

(q_0) Gdy w stanie q_0 głowica czyta jedynkę, maszyna zostawia ją na taśmie, przesuwa głowicę w prawo i przechodzi do stanu q_1 zgodnie z $\delta(q_0, 1) = (1, R, q_1)$; jeżeli przeczyta blank, pozostawia go, nie rusza głowicy i przechodzi do stanu q_A (słowo było parzyste) zgodnie z $\delta(q_0, \sqcup) = (\sqcup, -, q_A)$.

(q_1) Gdy w stanie q_1 głowica przeczyta kolejną jedynkę, pozostawi ją na taśmie, przesunie głowicę w prawo i przejdzie do stanu q_0; jeżeli przeczyta blank, pozostawia go, nie rusza głowicy i przechodzi do stanu q_R (słowo było nieparzyste) zgodnie z $\delta(q_1, \sqcup) = (\sqcup, -, q_R)$.

Program maszyny cyklicznie zmienia stany q_0 i q_1, dopóki głowica maszyny nie napotka blanka. Jeżeli pierwszy blank pojawi się, gdy program jest w stanie q_0, to maszyna akceptuje słowo; gdy jest w stanie q_1, to je odrzuca. ◁

Turing wykazał, że jeśli algorytm jest czymś, co może obliczyć maszyna Turinga, to Entscheidungsproblem jest nierozstrzygalny. Dał też argument za czymś więcej: uzasadniał, że jeśli coś może być policzone przez osobę wykonującą sekwencję instrukcji (co stanowi dla nas intuicję *obliczalności*), to może być obliczone przez maszynę Turinga.

Hipoteza 1.1.1 (Churcha–Turinga)**.** *Każdy problem, który może być intuicyjnie uznany za obliczalny, jest rozwiązywalny przez maszynę Turinga.*

Maszyna Turinga jest zatem najsilniejszym *modelem obliczeń*. W praktyce używa się też innych modeli obliczeń: niektóre są równoważne maszynie Turinga, inne — nie. To, którego używamy, zależy od tego, jaki problem rozważamy i w jakim aspekcie model obliczeń (z natury przybliżony) powinien być dokładny.

System kodowania liczb, który zapisuje je jako ciągi jedynek (podobnie jak w przykładzie), nazywany jest *systemem unarnym* (*jedynkowym*):

$$\underbrace{1 \ldots 1}_{n} = \mathbf{n}$$

W ten sposób możemy kodować nie tylko liczby, ale dowolne, skończone obiekty. Podstawową wadą tego systemu jest czas potrzebny do zapisania liczb. Znacznie lepszym pomysłem jest *system binarny* (*dwójkowy*). Wartość liczby zapisaną w systemie dwójkowym jako $(n_k \ldots n_1 n_0)_2$ oblicza się tak, jak w systemie dziesiętnym, z tą różnicą, że za podstawę przyjmujemy 2, a nie 10.

$$\begin{array}{cccccc} (n_k & n_{k-1} & \ldots & n_1 & n_0)_2 \\ 2^k & 2^{k-1} & \ldots & 2^1 & 2^0 \end{array} \quad = \quad n_k \cdot 2^k + n_{k-1} \cdot 2^{k-1} + \ldots + n_1 \cdot 2^1 + n_0 \cdot 2^0 \quad = \quad (\mathbf{n})_{10}$$

Przykład 2.

$$\begin{array}{cccccc} (\mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0})_2 \\ 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array} \quad = \quad \mathbf{1} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{0} \cdot 2^3 + \mathbf{1} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{0} \cdot 2^0 \quad = \quad (37)_{10}$$

◁

W systemie dwójkowym pamięć potrzebna do zapisania liczby n wynosi $\lceil \log n \rceil$, podczas gdy w systemie jedynkowym — aż n (przez $\lceil x \rceil$ oznaczamy najmniejszą liczbę całkowitą nie mniejszą niż x, zaś log oznacza logarytm przy podstawie 2).

n	unarnie	binarnie
1	1	1
2	11	10
3	111	11
4	1111	100
5	11111	101
6	111111	110
7	1111111	111

W przykładzie z algorytmem Euklidesa pojawił się problem czasu działania algorytmu. W pewnych zagadnieniach niezwykle istotna jest zużyta pamięć, która ma oczywisty związek z czasem: odwołanie do pamięci (zapis/odczyt danych) wymaga czasu! Oszacowaniem wielkości zużywanych zasobów zajmuje się *teoria złożoności*.

Jednym z przełomowych niegdyś odkryć było stwierdzenie, że dwie dowolne maszyny, których hardware osiągnął dostateczny poziom złożoności i elastyczności, są sobie równoważne. Równoważność polega na tym, że dla dowolnych dwóch maszyn A i B można napisać programy takie, że dla dowolnych danych wejściowych maszyny te zwrócą ten sam wynik. Zatem wszystkie współczesne komputery ogólnego przeznaczenia są faktycznie *uniwersalnymi maszynami Turinga* i są sobie równoważne. Ale nie wszystko można policzyć. Oceną tego, czy coś można policzyć, zajmuje się dział zwany *obliczalnością*.

Podsumowując:

- Nawet najszybsze komputery nie poradzą sobie ze źle napisanymi programami. Zawsze warto zastanowić się nad tym, jaki jest koszt działania algorytmu i czy nie można go zmniejszyć.
- Czasami algorytm po prostu nie istnieje, a rozważany problem może okazać się wyjątkowo trudny. Jednak warto mieć rozeznanie, co już wiadomo na temat problemów nierozwiązywalnych, aby niepotrzebnie „nie wyważać otwartych drzwi". *

Rozdział 2

Teoria języków i automatów

2.1 Podstawowe pojęcia

Język programowania musi mieć jednoznaczną składnię i jednoznaczną semantykę. Nie możemy sobie pozwolić na niejednoznaczność, np.

Kup 5 jajek, a jeśli będą bułki, to kup 10.

W odróżnieniu od języka naturalnego, języki programowania są tworami sztucznymi. Dla języków naturalnych praktycznie nie jest możliwe stworzenie dokładnej gramatyki. W przypadku języków programowania jest to konieczne: określa się dla nich gramatykę i za poprawne uznaje się tylko to, co z tej gramatyki można wyprowadzić.

Aby mówić o językach i gramatykach, potrzebujemy pewnych podstawowych definicji.

- *Alfabet* to skończony, niepusty zbiór symboli (liter).
- *Słowo* jest skończonym ciągiem zestawionych razem symboli. Długość słowa w najczęściej oznaczamy przez $|w|$; słowo puste oznaczamy symbolem ε.
- *Prefiksem* (*przedrostkiem*) słowa jest dowolna liczba symboli rozpoczynających to słowo.
- *Sufiksem* (*przyrostkiem*) słowa nazywamy dowolną liczbę symboli kończących słowo.
- *Konkatenacją* (*złożeniem*) słów u i v jest słowo powstałe poprzez sklejenie tych słów; piszemy $u \circ v$, $u \cdot v$ lub po prostu uv. Wielokrotną konkatenację zapisujemy jako indeks górny, czyli

$$u^i = \underbrace{u \circ \ldots \circ u}_{i},$$

  gdzie u jest dowolnym słowem.
- *Językiem formalnym* nazywamy zbiór słów złożonych z symboli jakiegoś ustalonego alfabetu.

2.1.1 Przykłady słów, prefiksy, sufiksy

- Niech alfabet $\mathcal{A}$ składa się z symboli a, b, c, d; piszemy wówczas $\mathcal{A} = \{a, b, c, d\}$.
- Przykładowe słowa nad powyższym alfabetem $\mathcal{A}$ to ε, a, $abcbabb$.
- Prefiksy słowa $abcba$ to $\varepsilon, a, ab, abc, abcb, abcba$; a sufiksy to $\varepsilon, a, ba, cba, bcba, abcba$.
- Słowo puste ε jest elementem neutralnym dla operatora konkatenacji, tj. $\varepsilon w = w\varepsilon = w$ dla dowolnego słowa w.

2.1.2 Przykłady języków formalnych

Kiedy mówimy o języku, najczęściej jego alfabet oznaczamy symbolem Σ. Załóżmy zatem, że $\Sigma = \{a, b\}$.

- Zarówno zbiór pusty $\varnothing$, jak i zbiór $\{\varepsilon\}$ są językami nad dowolnym alfabetem, a więc także nad ustalonym alfabetem Σ.
- Zbiór wszystkich słów nad Σ jest językiem i oznaczamy go jako Σ^*. W naszym przypadku *

$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\}.$$

  Zauważmy, że Σ^* z operacją konkatenacji jest monoidem, tzn. konkatenacja jest tu łączna i ma element neutralny.

- Przez Σ^+ oznaczamy zbiór wszystkich słów nad Σ bez słowa pustego: *

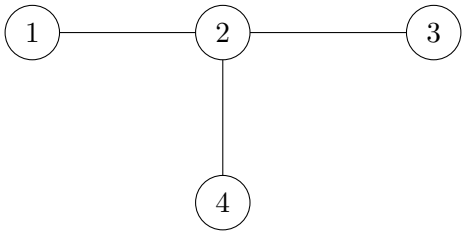
$$\Sigma^+ = \Sigma^* \setminus \{\varepsilon\} = \{a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\}.$$

  To także jest oczywiście język nad Σ.

- Dowolny — skończony lub nieskończony — podzbiór zbioru Σ^* jest językiem nad alfabetem Σ. *

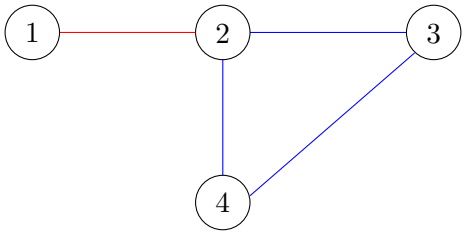
2.1.3 Grafy

Definicja 2.1.1. *Grafem nieskierowanym* nazywamy parę (V, E), gdzie V jest zbiorem *wierzchołków*, E jest zbiorem *krawędzi*, zaś każda krawędź jest nieuporządkowaną parą wierzchołków[1], czyli $E \subseteq \{\{u, v\} \mid u, v \in V\}$. Mówimy ponadto, że graf ten jest skończony, jeśli V jest skończony. *



Rysunek 2.1: Przykład grafu nieskierowanego, w którym $V = \{1, 2, 3, 4\}$, zaś $E = \{\{1, 2\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$

Definicja 2.1.2. *Ścieżką* w grafie nieskierowanym $G = (V, E)$ nazywamy ciąg wierzchołków $(v_1, v_2, \ldots, v_k)$ taki, że $v_i \in V$ dla $i = 1, \ldots, k$ oraz $\{v_i, v_{i+1}\} \in E$ dla $i = 1, \ldots, k-1$. Jeśli $v_1 = v_k$, to ścieżkę nazywamy *cyklem*. *Długość ścieżki* to liczba krawędzi na tej ścieżce, czyli $k - 1$.

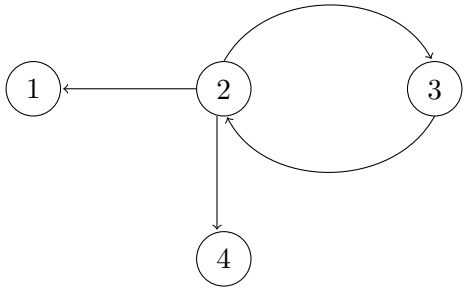


Rysunek 2.2: Na czerwono zaznaczono ścieżkę $(1, 2)$; niebieska ścieżka $(2, 4, 3, 2)$ jest cyklem

Definicja 2.1.3. *Grafem skierowanym* nazywamy parę (V, E), gdzie V jest zbiorem *wierzchołków*, E jest zbiorem *krawędzi*, zaś każda krawędź jest uporządkowaną parą wierzchołków, czyli $E \subseteq \{(u, v) \mid u, v \in V\}$. Mówimy, że graf ten jest skończony, jeśli V jest skończony.

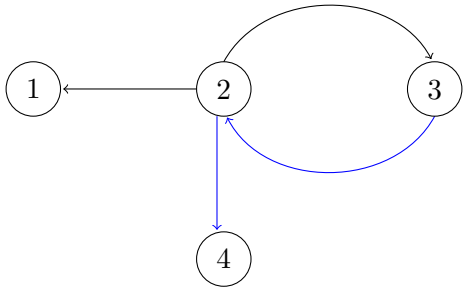
[1] Dopuszczamy też zdegenerowaną „parę” jednoelementową.

Zamiast $(u, v) \in E$ piszemy często $u \to v \in E$ lub po prostu $u \to v$. O krawędzi $u \to v$ mówimy, że *wychodzi* z u i *dochodzi* do v.



Rysunek 2.3: Przykład grafu skierowanego, w którym $V = \{1, 2, 3, 4\}$, zaś $E = \{(2, 1), (2, 3), (2, 4), (3, 2)\}$

Definicja 2.1.4. *Ścieżką* w grafie skierowanym $G = (V, E)$ nazywamy ciąg wierzchołków $(v_1, v_2, \ldots, v_k)$ taki, że $v_i \in V$ dla $i = 1, \ldots, k$ oraz $(v_i, v_{i+1}) \in E$ dla $i = 1, \ldots, k - 1$.



Rysunek 2.4: W grafie istnieje (niebieska) ścieżka $3 \to 2 \to 4$, ale nie istnieje ścieżka z 1 do 4

2.1.4 Drzewa

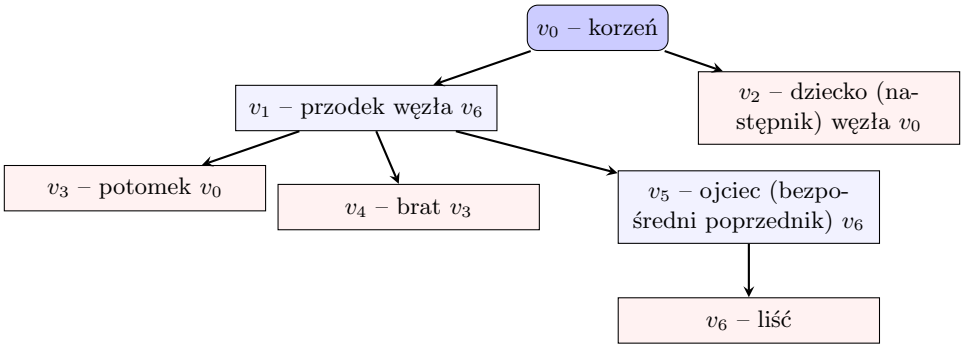
Poniższa definicja określa drzewo tak, jak zazwyczaj widzą je informatycy, przede wszystkim w kontekście używanych powszechnie struktur danych. Warto zaznaczyć, że klasyczna definicja drzewa w teorii grafów jest nieco inna.

Definicja 2.1.5. *Drzewem* nazywamy graf skierowany o następujących własnościach:

- istnieje dokładnie jeden wierzchołek (zwany *korzeniem*), do którego nie dochodzi żadna krawędź i z którego istnieje ścieżka do każdego innego wierzchołka;
- do każdego wierzchołka, nie będącego korzeniem, dochodzi dokładnie jedna krawędź.

Jako szczególny, zdegenerowany przypadek drzewa dopuszczamy zazwyczaj graf pusty, czyli $(\varnothing, \varnothing)$.

Wierzchołki drzewa często nazywamy *węzłami*. Wierzchołek, z którego nie wychodzi żadna krawędź, nazywamy *liściem*. Wierzchołek, który nie jest liściem, nazywamy *węzłem wewnętrznym*. Jeżeli krawędź $u \to v$ należy do drzewa, to mówimy, że u jest *poprzednikiem* v, zaś v jest *następnikiem* (lub *dzieckiem*) u. Czasami korzysta się z dalszych relacji rodzinnych — zob. rysunek 2.5.


v_0 – korzeń
v_1 – przodek węzła v_6
v_2 – dziecko (następnik) węzła v_0
v_3 – potomek v_0
v_4 – brat v_3
v_5 – ojciec (bezpośredni poprzednik) v_6
v_6 – liść

Rysunek 2.5: Przykładowe nazwy relacji w drzewie; węzły wewnętrzne są fioletowe, zaś liście — różowe.

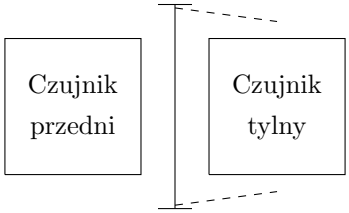
Ze względu na zastosowania drzew często przyjmujemy, że następniki każdego wierzchołka są w pewnien sposób uporządkowane; uprawnione jest zatem mówienie o „lewym następniku", „drugim następniku", „skrajnie prawym następniku" itp.

2.2 Języki regularne

2.2.1 Deterministyczny automat skończony (DFA)

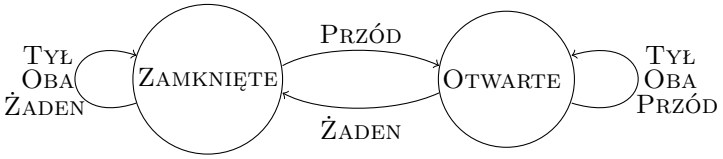
Automaty skończone to najprostsze modele, które co prawda nie nadają się do symulowania komputerów do ogólnych obliczeń, ale wykorzystywane są w przetwarzaniu tekstów, konstrukcji kompilatorów i projektowaniu sprzętu.

Przykład 3 (zob. [8])**.** Mechanizm kontrolujący otwieranie automatycznych jednokierunkowych drzwi wahadłowych jest przykładem urządzenia, którego działanie można opisać za pomocą automatu skończonego.



Drzwi otwierają się tylko wówczas, jeśli przedni czujnik wykryje osobę, a w zasięgu tylnego czujnika nikt nie stoi. Drzwi otwierają się zawsze w kierunku tylnego czujnika. Urządzenie kontrolujące jest w jednym z dwóch stanów: Otwarte albo Zamknięte. Musi reagować na pojawienie się osoby na jednym z paneli:

- Przód — osoba jest w zasięgu tylko przedniego czujnika, więc należy otworzyć drzwi;
- Tył — osoba jest w zasięgu tylko tylnego czujnika, więc nie należy zamykać drzwi, aby nie uderzyć tej osoby; jeśli drzwi były zamknięte, to nie ma powodu ich otwierać;
- Oba — nie należy nic robić, aby nie uderzyć osoby za drzwiami;
- Żaden — nikt nie stoi na panelach, więc można zamknąć drzwi, jeżeli były otwarte.



Rysunek 2.6: Diagram stanów mechanizmu kontrolującego automatyczne drzwi wahadłowe.

◁

Urządzenie sterujące ruchem drzwi wahadłowych jest komputerem, któremu wystarcza jeden bit pamięci do zapisania stanu urządzenia: OTWARTE – ZAMKNIĘTE. Urządzenia takie jak windy, zmywarki lub automaty z kawą są nieco bardziej skomplikowane, ale wciąż do sterowania nimi wystarczają komputery o bardzo ograniczonej pamięci. Często ich działanie można opisać przy pomocy automatów skończonych.

Definicja 2.2.1 (Automat skończony lub inaczej automat skończenie stanowy)**.** *Deterministycznym automatem skończonym* nazywamy piątkę $(Q, \Sigma, \delta, q_0, F)$, gdzie:

Q — skończony zbiór stanów

Σ — skończony alfabet wejściowy

δ — funkcja przejścia $\delta\colon Q \times \Sigma \to Q$

q_0 — stan początkowy

F — zbiór stanów końcowych (akceptujących).

Tak jak dla maszyn Turinga reprezentowaliśmy funkcję przejścia w postaci grafu, tak z każdym automatem skończonym wiążemy *graf przejść*. W przykładzie 3 rysunek 2.6 przedstawiał funkcję przejścia automatu opisującego działanie drzwi wahadłowych. Funkcję przejścia, tak jak w przypadku maszyny Turinga, możemy przedstawić w postaci tabelki:

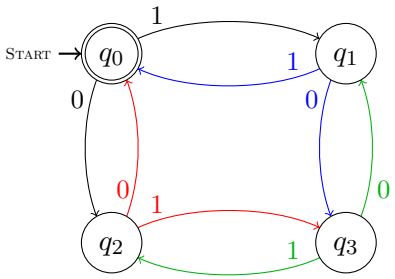
STAN	PRZÓD	TYŁ	OBA	ŻADEN
ZAMKNIĘTE	OTWARTE	ZAMKNIĘTE	ZAMKNIĘTE	ZAMKNIĘTE
OTWARTE	OTWARTE	OTWARTE	OTWARTE	ZAMKNIĘTE

Definicja 2.2.2. Niech $\mathbf{A} = (Q, \Sigma, \delta, q_0, F)$ będzie automatem skończonym, a $w = w_1 \dots w_n$ słowem nad alfabetem Σ. Automat $\mathbf{A}$ *akceptuje* słowo w, jeżeli istnieje ciąg stanów $(p_0, p_1, \dots, p_n)$ należących do Q taki, że

1. $p_0 = q_0$, czyli rozpoczynamy w stanie początkowym;
2. $\delta(p_i, w_i) = p_{i+1}$, czyli działamy zgodnie z funkcją przejścia;
3. $p_n \in F$, czyli kończymy w stanie końcowym.

Mówimy, że $\mathbf{A}$ *rozpoznaje język* L, jeżeli $L = \{w\colon \mathbf{A} \text{ akceptuje } w\}$. Zapisujemy to jako $L = L(\mathbf{A})$.

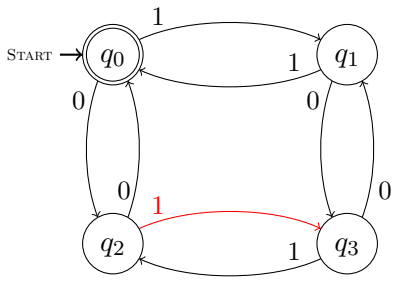
Przykład 4. Niech $\mathbf{A} = (Q, \Sigma, \delta, q_0, F)$ będzie automatem skończonym, gdzie $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$, a funkcja przejścia pokazana jest na rysunku poniżej.



Rysunek 2.7: Diagram przejść dla automatu skończonego A.

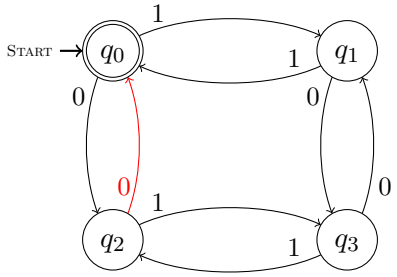
Działanie automatu $\mathbf{A}$ na słowie 1101.

$$\begin{aligned}\delta(q_0, 1101) &= q_1\\ \delta(q_1, 101) &= q_0\\ \delta(q_0, 01) &= q_2\\ \delta(q_2, 1) &= q_3\end{aligned}$$



Automat zatrzymuje się w stanie q_3, który nie jest stanem końcowym, zatem słowo 1101 nie zostało zaakceptowane. Zobaczmy, jak wygląda działanie automatu **A** na słowie 1010.

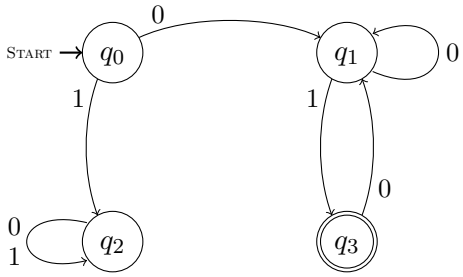
$$\begin{aligned}\delta(q_0, 1010) &= q_1\\ \delta(q_1, 010) &= q_3\\ \delta(q_3, 10) &= q_2\\ \delta(q_2, 0) &= q_0\end{aligned}$$



Automat zatrzymał się w stanie q_0, który jest stanem końcowym, zatem słowo 1010 zostało zaakceptowane.

Jeżeli dokładnie przeanalizujemy funkcję przejścia automatu **A**, to przekonamy się, że akceptuje on słowo wtedy i tylko wtedy, gdy ma ono parzystą liczbę zer i parzystą liczbę jedynek. ◁

Przykład 5. Automat **B**, którego funkcja przejścia przedstawiona jest na rysunku 2.8, akceptuje wszystkie słowa, które rozpoczynają się od 0 i kończą 1.



Rysunek 2.8: Diagram przejść dla automatu skończonego B.

◁

Wiemy już, co to jest automat skończony, potrafimy go formalnie zapisać, ale jak moglibyśmy opisać słowa akceptowane przez automat? Jednym ze sposobów są *wyrażenia regularne*. Na przykład

$$L(\mathbf{B}) = 0(0+1)^*1.$$

Ale o tym powiemy za chwilę...

Definicja 2.2.3. Język L nazywamy *językiem regularnym*, jeżeli istnieje deterministyczny automat skończony **A**, który go rozpoznaje, czyli $L = L(\mathbf{A})$.

2.2.2 Operacje regularne

W arytmetyce podstawowymi obiektami są liczby, a narzędziami, za pomocą których na nich operujemy, są różne operacje arytmetyczne, np. + (dodawanie) i · (mnożenie). W teorii obliczeń obiektami są języki, a operacjami — działania na tych językach. Trzy operacje: sumy, konkatenacji i gwiazdki Kleene'ego są operacjami regularnymi.

Definicja 2.2.4. Niech Σ będzie skończonym alfabetem i niech L_1, L_2 będą językami nad Σ.
Sumą języków L_1 i L_2 nazywamy $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$, czyli zwykłą sumę zbiorów.
Konkatenacją (złożeniem) języków L_1 i L_2 nazywamy $L_1L_2 = \{xy \mid x \in L_1,\ y \in L_2\}$.

Przykład 6. Niech $\Sigma = \{0, 1\}$.

1. Niech $L_1 = \{011, 11\}$, $L_2 = \{10, 1\}$. Wówczas

$$\begin{aligned} L_1L_2 &= \{01110, 1110, 0111, 111\}, \\ L_2L_1 &= \{10011, 1011, 1011, 111\} \\ &= \{10011, 1011, 111\}. \end{aligned}$$

2. Niech $L_1 = \{\varepsilon, 0, 00, 000, \ldots\}$, $L_2 = \{10, 1\}$. Wówczas

$$\begin{aligned} L_1L_2 &= \{\varepsilon 10, \varepsilon 1, 010, 01, 0010, 001, \ldots\} \\ &= \{10, 1, 010, 01, 0010, 001, \ldots\}, \\ L_2L_1 &= \{10\varepsilon, 1\varepsilon, 100, 10, 1000, 100, \ldots\} \\ &= \{1, 10, 100, 1000, \ldots\}. \end{aligned}$$

◁

Niech L będzie językiem nad alfabetem Σ. Wprowadźmy oznaczenie:

- $L^0 = \{\varepsilon\}$
- $L^i = LL^{i-1}$ dla $i \geqslant 1$.

Definicja 2.2.5. *Domknięciem* (lub *gwiazdką Kleene'ego*) języka L nazywamy zbiór

$$L^* = \bigcup_{i=0}^{\infty} L^i,$$

a *domknięciem dodatnim* zbiór

$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

Przykład 7. Niech $L = \{10, 11\}$. Wówczas

$$\begin{aligned} L^0 &= \{\varepsilon\}, \\ L^1 &= LL^0 = L, \\ L^2 &= LL = \{1010, 1011, 1110, 1111\}, \\ &\ldots \\ L^* &= L^0 \cup L^1 \cup L^2 \cup \ldots = \{\varepsilon, 10, 11, 1010, 1011, 1110, 1111, \ldots\}, \\ L^+ &= L^1 \cup L^2 \cup \ldots = \{10, 11, 1010, 1011, 1110, 1111, \ldots\}. \end{aligned}$$

Jak widać, $L^* = L^+ \cup \{\varepsilon\}$. ◁

Zauważmy, że suma i konkatenacja są operacjami binarnymi, a gwiazdka Kleene'ego jest operacją unarną.

Definicja 2.2.6. Zbiór A jest *zamknięty ze względu na* pewną operację $\circ$, jeżeli zastosowanie tej operacji do elementów tego zbioru daje w wyniku element ze zbioru, czyli

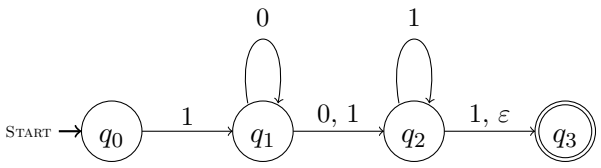
$$\forall x, y \in A\colon x \circ y \in A.$$

Okazuje się, że klasa języków regularnych jest zamknięta ze względu na sumę, konkatenację i gwiazdkę Kleene'ego. Aby udowodnić ten fakt, wygodnie jest wprowadzić nowe pojęcie *niedeterministycznego automatu skończonego*.

2.2.3 Niedeterministyczny automat skończony (NFA)

Maszyna deterministyczna w każdym stanie ma co najwyżej jedną możliwość ruchu (w przypadku automatu skończonego dokładnie jedną). Natomiast maszyna niedeterministyczna może mieć do wyboru więcej stanów, do których mogłaby przejść w kolejnym kroku.

Przykład 8. Niedeterminizm jest uogólnieniem determinizmu, zatem każdy deterministyczny automat skończony jest zarazem niedeterministycznym automatem skończonym, ale nie na odwrót.



Rysunek 2.9: Przykład niedeterministycznego automatu skończonego.

◁

W DFA każdy stan ma dokładnie jedno przejście dla każdego symbolu z alfabetu. W NFA z każdego stanu może wychodzić zero, jedna lub więcej strzałek i strzałki mogą być etykietowane przez symbole z alfabetu lub słowo puste ε. Sens przejścia pod wpływem słowa pustego stanie się jasny za chwilę, gdy formalnie zdefiniujemy NFA. *

Przypomnijmy: dla zbioru X przez $\mathcal{P}(X)$ oznaczamy *zbiór potęgowy* X, czyli zbiór wszystkich jego podzbiorów. Na przykład dla $X = \{1, 2\}$ jest $\mathcal{P}(X) = \{\varnothing, \{1\}, \{2\}, \{1, 2\}\}$. Dla dowolnego alfabetu Σ przez Σ_ε oznaczamy $\Sigma \cup \{\varepsilon\}$.

Definicja 2.2.7. *Niedeterministycznym automatem skończonym* nazywamy piątkę $(Q, \Sigma, \delta, q_0, F)$, gdzie

- Q — skończony zbiór stanów
- Σ — skończony alfabet wejściowy
- δ — funkcja przejścia $\delta\colon Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$
- q_0 — stan początkowy
- F — zbiór stanów końcowych (akceptujących). *

Definicja 2.2.8. Niech $\mathbf{A} = (Q, \Sigma, \delta, q_0, F)$ będzie niedeterministycznym automatem skończonym, a w słowem nad alfabetem Σ. Automat $\mathbf{A}$ *akceptuje* słowo w, jeżeli istnieje rozkład tego słowa nad Σ_ε, $w = w_1 \dots w_n$ ($w_i \in \Sigma_\varepsilon$), oraz ciąg stanów $(p_0, p_1, \dots, p_n)$ należących do Q taki, że

1. $p_0 = q_0$, czyli rozpoczynamy w stanie początkowym;
2. $\delta(p_i, w_i) \ni p_{i+1}$, czyli działamy zgodnie z funkcją przejścia;

3. $p_n \in F$, czyli kończymy w stanie końcowym.

Zwróćmy uwagę na szczególne cechy automatu niedeterministycznego. Po pierwsze, kolejnym stanem w ciągu stanów może być dowolny element zbioru wskazanego przez wartość funkcji przejścia. Po drugie, automat może wykonywać przejścia pod wpływem słowa pustego (nieformalnie: zamiast przeczytać kolejny symbol i wykonać przejście przezeń wyznaczone, automat może po prostu wykonać przejście wyznaczone przez słowo puste; sens rozkładu słowa wejściowego na symbole z alfabetu Σ i dowolnie rozmieszczone słowa puste jest właśnie taki). Zazwyczaj zatem dla danego słowa można wyznaczyć wiele ciągów stanów idących od stanu początkowego. By zaakceptować słowo, wystarczy że jeden z nich prowadzi do stanu końcowego. Dodajmy jeszcze, że opisanego tu niedeterminizmu nie należy wiązać z jakąkolwiek losowością; jest to jedynie sposób opisania alternatywnych ścieżek wyboru.

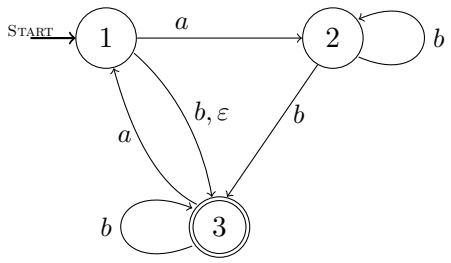
Choć wydaje się to zaskakujące, języki rozpoznawane przez DFA tworzą tę samą klasę co języki rozpoznawane przez NFA. Niedeterminizm nie wnosi zatem dodatkowych możliwości obliczeniowych. Fakt ten opisuje twierdzenie 2.2.1.

Definicja 2.2.9. Dwie maszyny M_1, M_2 są *równoważne*, jeżeli rozpoznają ten sam język, czyli gdy $L(M_1) = L(M_2)$.

Twierdzenie 2.2.1. *Dla każdego NFA istnieje równoważny mu DFA.*

Ideę dowodu powyższego twierdzenia przedstawimy na przykładzie.

Przykład 9. Zobaczmy, w jaki sposób można przekształcić automat niedeterministyczny w równoważny mu automat deterministyczny. Dla przykładu rozważmy automat $N = (\{1, 2, 3\}, \{a, b\}, \delta, 1, \{3\})$, którego funkcja przejścia δ jest na rysunku poniżej.



Rysunek 2.10: Diagram przejść dla niedeterministycznego automatu N.

Automat deterministyczny $M = (Q', \{a, b\}, \delta', q_0', F')$ równoważny automatowi N będzie mieć co najwyżej 8 stanów:

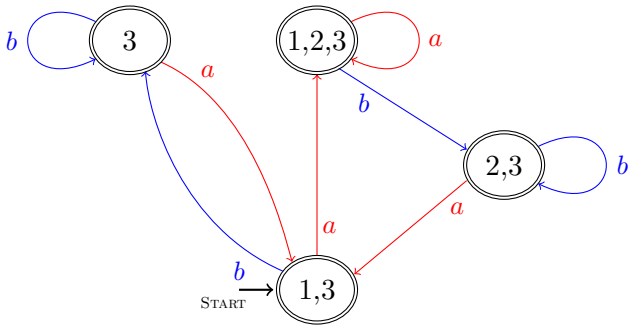
$$Q' = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Stan początkowy automatu deterministycznego to zbiór zawierający stan początkowy automatu N oraz wszystkie stany, do których można z niego przejść pod wpływem ε, czyli $q_0' = \{1, 3\}$. Nowe stany akceptujące to zbiory zawierające stan końcowy automatu N, czyli $F' = \{\{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Pozostaje opisać funkcję przejścia δ'. W tabelce poniżej kolorem szarym zaznaczone są przejścia nieużyteczne, czyli przejścia ze stanów, do których automat nigdy nie dotrze (np. stan $\{1\}$ jest nieosiągalny, gdyż na początku automat jest w stanie $\{1, 3\}$, a żadna wartość funkcji przejścia nie prowadzi do $\{1\}$). *

	a	b
$\{1\}$	$\{1,2\}$	$\{3\}$
$\{2\}$	$\varnothing$	$\{2,3\}$
$\{3\}$	$\{1,3\}$	$\{3\}$
$\{1,2\}$	$\{1,2\}$	$\{2,3\}$
$\{1,3\}$	$\{1,2,3\}$	$\{3\}$
$\{2,3\}$	$\{1,3\}$	$\{2,3\}$
$\{1,2,3\}$	$\{1,2,3\}$	$\{2,3\}$

Funkcję δ' opisuje również diagram na rysunku 2.11. Kolory mają jedynie za zadanie ułatwić czytanie diagramu.



Rysunek 2.11: Diagram przejść dla deterministycznego automatu skończonego M.

Widać, że skonstruowana maszyna M działa poprawnie: dla dowolnych danych wejściowych, w każdym kroku obliczeń maszyna M jest w stanie odpowiadającym podzbiorowi stanów N, w których może znaleźć się maszyna niedeterministyczna w analogicznym momencie. ◁

Wniosek 2.2.1. *Język jest regularny (zob. definicja 2.2.3) wtedy i tylko wtedy, gdy jest rozpoznawany przez pewien niedeterministyczny automat skończony.*

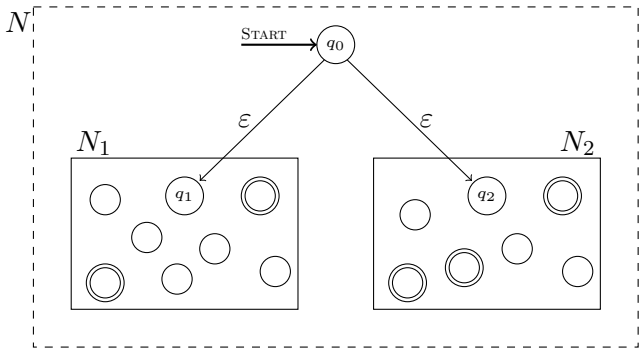
Aby pokazać, że jakiś język jest regularny, wystarczy skonstruować automat skończony (deterministyczny bądź niedeterministyczny), który go rozpoznaje.

2.2.4 Zamkniętość ze względu na operacje regularne

Powróćmy do tematu zamkniętości klasy języków regularnych ze względu na sumę, konkatenację i gwiazdkę Kleene'ego. W poniższych dowodach zakładamy, że rozważane automaty mają rozłączne zbiory stanów, a q_0 nie należy do żadnego z nich.

Twierdzenie 2.2.2. *Klasa języków regularnych jest zamknięta ze względu na sumę.*

Ideę dowodu prezentuje schemat z rysunku 2.12. *



Rysunek 2.12: $L(N) = L_1 \cup L_2$, gdzie $L_1 = L(N_1)$ oraz $L_2 = L(N_2)$.

*

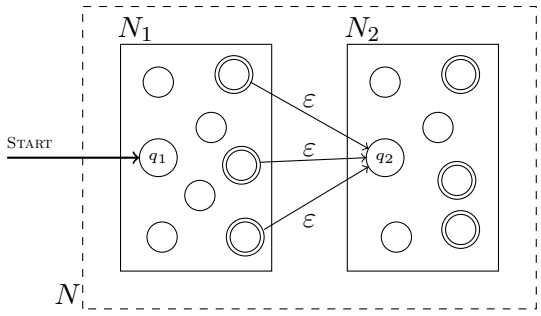
Dowód twierdzenia 2.2.2. Niech L_1 i L_2 będą językami regularnymi, czyli $L_1 = L(N_1)$ oraz $L_2 = L(N_2)$, gdzie $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ to niedeterministyczne automaty skończone. Automat rozpoznający sumę $L_1 \cup L_2$ to $N = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, \delta, q_0, F_1 \cup F_2)$, gdzie

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & q \in Q_1 \\ \delta_2(q,a), & q \in Q_2 \\ \{q_1, q_2\}, & q = q_0 \wedge a = \varepsilon \\ \varnothing, & q = q_0 \wedge a \neq \varepsilon. \end{cases}$$

□

Twierdzenie 2.2.3. *Klasa języków regularnych jest zamknięta ze względu na konkatenację.*

Ideę dowodu prezentuje schemat z rysunku 2.13.



Rysunek 2.13: $L(N) = L_1 \circ L_2$, gdzie $L_1 = L(N_1)$ oraz $L_2 = L(N_2)$.

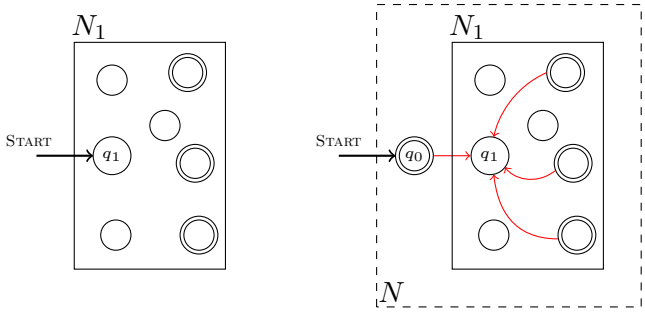
Dowód twierdzenia 2.2.3. Niech L_1 i L_2 będą językami regularnymi rozpoznawanymi przez niedeterministyczne automaty skończone, odpowiednio $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ i $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Automat rozpoznający konkatenację $L_1 \circ L_2$ to $N = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F_2)$, gdzie

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & q \in Q_1 \setminus F_1 \\ \delta_1(q,a), & q \in F_1 \wedge a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\}, & q \in F_1 \wedge a = \varepsilon \\ \delta_2(q,a), & q \in Q_2. \end{cases}$$

□

Twierdzenie 2.2.4. *Klasa języków regularnych jest zamknięta ze względu na gwiazdkę Kleene'ego.*

Ideę dowodu prezentuje schemat z rysunku 2.14. Zakładamy, że $L = L(N_1)$.



Rysunek 2.14: $L^* = L(N)$, gdzie $L = L(N_1)$; czerwone strzałki na rysunku są etykietowane przez ε.

Dowód twierdzenia 2.2.4. Niech L będzie językiem regularnym rozpoznawanym przez niedeterministyczny automaty skończone $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$. Automat rozpoznający L^* to $N =$

$(Q_1 \cup \{q_0\}, \Sigma, \delta, q_0, F_1 \cup \{q_0\})$, gdzie

$$\delta(q,a) = \begin{cases} \{q_1\}, & q = q_0 \wedge a = \varepsilon \\ \varnothing, & q = q_0 \wedge a \neq \varepsilon \\ \delta_1(q,a), & q \in Q_1 \setminus F_1 \\ \delta_1(q,a), & q \in F_1 \wedge a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\}, & q \in F_1 \wedge a = \varepsilon. \end{cases}$$

□

2.2.5 Wyrażenia regularne

Zdefiniujemy teraz pewnien sposób reprezentowania języków regularnych za pomocą napisów zwanych wyrażeniami regularnymi. Wyrażenia regularne są powszechnie wykorzystywane w wielu sytuacjach, w których trzeba zwięźle opisać wzorzec tekstu, np. wyszukując tekst w edytorze lub wydając bardziej złożone polecenie w systemie operacyjnym.

Definicja 2.2.10. Niech Σ będzie alfabetem. *Wyrażenia regularne* nad Σ i zbiory przez nie reprezentowane definiujemy rekurencyjnie w następujący sposób:

- $\varnothing$ jest wyrażeniem regularnym reprezentującym zbiór pusty,
- ε jest wyrażeniem regularnym reprezentującym zbiór $\{\varepsilon\}$,
- dla każdego $a \in \Sigma$, a jest wyrażeniem regularnym reprezentującym zbiór $\{a\}$,
- jeżeli r i s są wyrażeniami regularnymi reprezentującymi języki R i S, to napisy $(r+s)$, (rs) i (r^*) są wyrażeniami regularnymi reprezentującymi, odpowiednio, zbiory $R \cup S$, RS i R^*.

Używamy ponadto zapisu (r^+), który reprezentuje język R^+ (zob. definicja 2.2.5).

Przy zapisywaniu wyrażeń regularnych opuszczamy nawiasy zgodnie z założeniem, że $*$ ma wyższy priorytet niż złożenie i $+$, np. wyrażenie $((0(1^*))+0)$ możemy zapisać jako 01^*+0 (reprezentuje ono zbiór $\{0, 01, 011, 0111 \ldots\}$).

Prawdziwe są też różne tożsamości dotyczące wyrażeń regularnych, np. $(rs)t = r(st)$, $(r+s)t = rt + st$, $rr^* = r^+$.

Przykład 10.

wyrażenie regularne	reprezentowany zbiór
00	$\{00\}$
$(0+1)^*$	zbiór wszystkich ciągów złożonych z 0 i 1
$(0+1)^*00(0+1)^*$	zbiór wszystkich ciągów złożonych z 0 i 1, w których występują dwa zera „pod rząd”
$(1+10)^*$	wszystkie słowa zaczynające się od 1 i nie zawierające dwóch kolejnych zer
$(\varepsilon+00)1^*$	dowolny ciąg jedynek bądź dowolny ciąg jedynek rozpoczynający się ciągiem 00

◁

Twierdzenie 2.2.5. *Język jest regularny wtedy i tylko wtedy, gdy jest reprezentowany przez wyrażenie regularne.*

Aby udowodnić twierdzenie 2.2.5, należy udowodnić dwa poniższe lematy (implikacje w dwie strony).

Lemat 2.2.1. *Jeżeli język jest reprezentowany przez wyrażenie regularne, to jest regularny.*

Dowód. Niech r będzie wyrażeniem regularnym opisującym pewien język R. Dowód jest indukcyjny ze względu na budowę wyrażenia r. Rozważmy zatem wszystkie przypadki postaci wyrażenia r podane w definicji 2.2.10. Dla każdego języka R opisanego przez r podamy NFA akceptujący ten język. Ponieważ dla każdego NFA istnieje równoważny mu DFA, zatem będzie to dowód, że R jest regularny.

- $r = \varnothing$ opisuje język pusty, zatem $N_\varnothing = (\{q_0\}, \Sigma, \delta, q_0, \varnothing)$, gdzie $\delta(q_0, x) = \varnothing$ dla dowolnego symbolu z Σ;
- $r = \{\varepsilon\}$ opisuje język $L(R) = \{\varepsilon\}$, zatem akceptujący go NFA to $N_\varepsilon = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$, gdzie $\delta(q_0, x) = \varnothing$ dla dowolnego symbolu z Σ;
- $r = \{a\}$, gdzie a jest dowolnym symbolem z Σ; R opisuje język $L(R) = \{a\}$, zatem akceptujący go NFA to $N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$, gdzie

$$\delta(q_0, x) = \begin{cases} \{q_1\}, & x = a \\ \varnothing, & x \neq a; \end{cases}$$

- załóżmy, że s i t są wyrażeniam regularnymi reprezentującymi, odpowiednio, języki regularne S i T; wówczas z twierdzeń o zamkniętości klasy języków regularnych na operacje regularne (tj. $\cup$, $\circ$ i *) wyrażenia $r = s + t$, $r = st$ i $r = s^*$ opisują języki regularne. □

Lemat 2.2.2. *Jeżeli język jest regularny, to jest reprezentowany przez wyrażenie regularne.*

Dowód. [zob. [8], str. 72] □

2.3 Języki nieregularne

Jak można się spodziewać, nie wszystkie języki są regularne. Na przykład język $\{0^n 1^n : n \in \mathbb{N}\}$ nie jest regularny. Wytłumaczenie wydaje się proste: automat musiałby najpierw „zliczyć" liczbę zer, aby potem móc zaakceptować taką samą liczbę jedynek. Ponieważ zer może być dowolnie wiele, to automat musiałby mieć nieskończenie wiele stanów. I chociaż w tym wypadku tak jest (język nie jest regularny), to jednak intuicja nie zawsze jest prawdziwa i taki fakt wymaga precyzyjnego dowodu.

Przykład 11. Zdefiniujmy dwa języki nad alfabetem $\{0, 1\}$:

- $A = \{w : w$ ma jednakową liczbę symboli 0 i 1$\}$,
- $B = \{w : w$ ma jednakową liczbę wystąpień podciągów 01 i 10$\}$. *

Okazuje się, że A nie jest regularny, podczas gdy B — jest (dowód regularności języka B zostawiamy jako proste ćwiczenie). ◁

Aby uzasadnić, że język jest regularny, wystarczy napisać opisujące go wyrażenie regularne lub skonstruować akceptujący go automat skończony (deterministyczny bądź niedeterministyczny). Natomiast, aby udowodnić, że jakiś język nie jest regularny, zwykle stosujemy tzw. *lemat o pompowaniu.*

*

Twierdzenie 2.3.1 (Lemat o pompowaniu)**.** *Jeżeli A jest językiem regularnym, to istnieje taka liczba p, że dla każdego słowa $s \in A$ o długości co najmniej p istnieje podział tego słowa na trzy części $s = xyz$ spełniający następujące warunki:*

1. $\forall i \geqslant 0 : xy^i z \in A$;
2. $|y| > 0$;
3. $|xy| \leqslant p$,

gdzie $|w|$ oznacza długość słowa w. □

Istotą powyższego twierdzenia jest to, że w języku regularnym każde odpowiednio długie słowo musi zawierać fragment y dający się wielokrotnie powielić („napompować"). Jeżeli zatem wykażemy, że dla dowolnej długości słowa nie ma w nim „pompowalnego" fragmentu, to język nie jest regularny.

2.3.1 Gramatyki bezkontekstowe i kontekstowe

Gramatyki bezkontekstowe

Automaty skończone i języki regularne to najprostszy opis języków. Nieco silniejszy sposób to opis przy użyciu gramatyk bezkontekstowych. Za pomocą takich gramatyk można opisywać rekurencyjnie własności języków. Stosuje się je powszechnie w specyfikacji i kompilacji języków programowania.

W przeciwieństwie do automatu, który opisuje sposób *akceptacji* słowa, gramatyka opisuje sposób, w jaki można *wygenerować* słowo należące do języka.

Definicja 2.3.1 (Gramatyka bezkontekstowa)**.** *Gramatykę bezkontekstową* nazywamy czwórkę $G = (N, T, P, S)$, gdzie:

N — skończony zbiór symboli *nieterminalnych* (zmiennych)

T — skończony zbiór symboli *terminalnych* (końcowych)

P — skończony zbiór *produkcji*; każda produkcja jest postaci $A \to \alpha$, gdzie $A \in N$, $\alpha \in (N \cup T)^*$

S — symbol startowy, $S \in N$.

Definicja 2.3.2 (Wyprowadzenie w gramatyce $G = (N, T, P, S)$)**.**

- Jeżeli $A \to \alpha \in P$ oraz $x, y \in (N \cup T)^*$, to

$$xAy \Rightarrow_G x\alpha y$$

  tzn. w gramatyce G słowo $x\alpha y$ jest *bezpośrednio wyprowadzalne* ze słowa xAy.

- Jeżeli $x_1, \ldots, x_m \in (N \cup T)^*$, $m \geqslant 1$, oraz

$$x_1 \Rightarrow_G x_2 \Rightarrow_G \ldots \Rightarrow_G x_m,$$

  piszemy $x_1 \Rightarrow_G^* x_m$ i mówimy, że x_m jest *wyprowadzalne* z x_1 w gramatyce G.

Definicja 2.3.3 (Język generowany przez gramatykę)**.** *Językiem generowanym* przez gramatykę $G = (N, T, P, S)$ jest zbiór

$$L(G) = \{w \mid w \in T^* \wedge S \Rightarrow_G^* w\}.$$

Czyli język $L(G)$ jest zbiorem wszystkich słów, które można wyprowadzić z symbolu startowego S i które złożone są z samych symboli terminalnych.

Przykład 12. Niech $G = (N, T, P, S)$ taka, że $N = \{S\}$, $T = \{a, b\}$, $P = \{S \to aSb, S \to ab\}$. Przykładowe wyprowadzenia w G:

$$\begin{aligned} S &\Rightarrow ab \\ S &\Rightarrow aSb \Rightarrow aabb \\ S &\Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb \\ S &\Rightarrow aSb \Rightarrow aaSbb \Rightarrow \ldots \Rightarrow a^{n-1}Sb^{n-1} \Rightarrow a^n b^n \end{aligned}$$

Zatem język generowany przez powyższą gramatykę to $L(G) = \{a^n b^n : n \geqslant 1\}$. ◁

Automaty to **akceptory** — rozpoznają (tzn. akceptują bądź nie) słowa z danego języka.

Gramatyki to **generatory** — generują słowa należące do języka.

Maszyny rozpoznające języki bezkontekstowe to *automaty ze stosem*. Automat ze stosem różni się od automatu skończonego (zob. definicja 2.2.1) posiadaniem dodatkowej taśmy — stosu. Pod wpływem czytanego symbolu, może nie tylko zmienić stan i zawartość taśmy wejściowej, ale może również zapisać słowo lub odczytać pojedynczy symbol ze stosu.

Gramatyki kontekstowe

Gramatyki kontekstowe różnią się od bezkontekstowych postacią produkcji.

Definicja 2.3.4 (Gramatyka kontekstowa)**.** Gramatyka $G = (N, T, P, S)$ jest *kontekstowa*, jeśli produkcje są postaci:

- $S \to \varepsilon$
- $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, gdzie $A \in N$, $\alpha_1, \alpha_2, \beta \in (N \cup T)^*$, $\beta \neq \varepsilon$, $S \notin w$.

Zatem w gramatyce kontekstowej symbol A może zostać zastąpiony słowem β tylko „w kontekście", tzn. gdy jest otoczony słowami α_1 i α_2 i kontekst ten musi zostać zachowany. Prawie każdy język, jaki możemy sobie wyobrazić, jest językiem kontekstowym — generowanym przez pewną gramatykę kontekstową. Jednakże „prawie" czyni wielką różnicę...

Dowody, że pewne języki nie są kontekstowe, oparte są na metodzie diagonalizacji — rodzaju dowodu nie wprost.

Przykład 13. Rozważmy gramatykę $G = (N, T, P, S)$ ze zbiorem terminali $T = \{a, b, c\}$, nieterminali $N = \{S, X, B\}$ oraz produkcji P:

$$\begin{array}{lll} S \to \varepsilon & X \to aBXc & Ba \to aB \\ S \to X & X \to abc & Bb \to bb \end{array}$$

Pojęcia gramatyki używamy tu w ogólniejszym znaczeniu, w którym produkcje mają swobodniejszą postać. Idea wyprowadzenia — część słowa, odpowiadającą lewej stronie produkcji, zastępujemy słowem odpowiadającym prawej stronie — jest oczywiście taka sama.

Język gramatyki G to:

$$L(G) = \{a^n b^n c^n : n \in \mathbb{N}\}.$$

Niestety gramatyka G nie jest kontekstowa — nie zachowuje bowiem kontekstu. Ale łatwo ją przekształcić na gramatykę kontekstową (kontekst, który musi zostać zachowany, zaznaczony jest na niebiesko; symbole z kreską to dodatkowe symbole):

$$Ba \to B\overline{a} \qquad B\overline{a} \to \overline{B}\overline{a} \qquad \overline{B}\overline{a} \to \overline{B}B \qquad \overline{B}B \to aB$$

Otrzymana w taki sposób gramatyka $\overline{G} = (\overline{N}, T, \overline{P}, S)$ jest równoważna gramatyce G, tzn. $L(G) = L(\overline{G})$. ◁

Można zatem postawić równoważną definicję gramatyki kontekstowej:

Definicja 2.3.5. Gramatyka $G = (N, T, P, S)$ jest *monotoniczna*, jeżeli produkcje są postaci:

- $S \to \varepsilon$
- $v \to u$, gdzie $|v| \leqslant |u|$, $S \notin u$.

Twierdzenie 2.3.2. *Gramatyki kontekstowe i monotoniczne generują te same języki.*

Maszynami, które rozpoznają języki kontekstowe są (niedeterministyczne) automaty liniowo ograniczone.

Automat liniowo ograniczony to ograniczona wersja maszyny Turinga (stąd inna nazwa *maszyna Turinga ograniczona liniowo*), która podczas obliczenia na słowie wejściowym długości n może wykorzystać jedynie n komórek taśmy. Innymi słowy, automat korzysta jedynie z pamięci pierwotnie zajętej przez słowo wejściowe. Działa niedeterministycznie.

Równość klas automatów liniowo ograniczonych deterministycznych i niedeterministycznych jest problemem otwartym.

Gramatyki nieograniczone

Gramatyki zwane *semigramatykami Thuego*, *gramatykami typu 0*, *strukturami frazowymi* lub *gramatykami nieograniczonymi* są najszerszą klasą gramatyk, w których produkcje są postaci

$$\alpha \to \beta,$$

gdzie α i β są dowolnymi słowami, przy czym $\alpha \neq \varepsilon$.

Gramatyki typu 0 generują języki akceptowane przez maszyny Turinga (niekoniecznie z własnością stopu). Maszyna posiada *własność stopu*, jeżeli zatrzymuje się dla każdego słowa wejściowego (akceptuje bądź odrzuca to słowo).

Definicja 2.3.6. Języki akceptowane przez maszyny Turinga z własnością stopu nazywamy *rekurencyjnymi* (jest to *klasa języków rekurencyjnych*).

Języki akceptowane przez maszyny Turinga bez własności stopu nazywamy *rekurencyjnie przeliczalnymi* (jest to *klasa języków rekurencyjnie przeliczalnych*).

2.4 Problemy decyzyjne i hierarchia Chomsky'ego

2.4.1 Przykłady problemów decyzyjnych

Ważną kwestią jest posiadanie algorytmów decyzyjnych odpowiadających na różne pytania. Interesujące mogą być np. pytania:

- Czy dany język jest pusty?
- Czy jest skończony, czy nie?
- Czy dany zbiór jest równy innemu?
- Czy dane słowo należy, czy też nie należy do jakiegoś języka?

2.4.2 Typy gramatyk — porównanie

Dla każdej klasy (innymi słowy zbioru) języków formalnych istnieje wiele równoważnych definicji. Można definiować klasę języków jako zbiór wszystkich języków generowanych przez pewien typ gramatyk (np. klasa języków bezkontekstowych, to zbiór wszystkich języków, dla których istnieje gramatyka bezkontestowa). Można też zdefiniować klasę języków jako zbiór wszystkich języków akceptowanych przez pewnien rodzaj maszyn (np. klasa języków regularnych to zbiór wszystkich języków akceptowanych przez automaty skończone). Klasa języków rekurencyjnie przeliczalnych to zbiór wszystkich języków generowanych przez gramatyki nieograniczone. Jest to ten sam zbiór co zbiór wszystkich języków akceptowanych przez maszyny Turinga. Ale zbiór ten można opisać na wiele innych sposobów: przy użyciu częściowych funkcji rekurencyjnych, za pomocą rachunku lambda, jako schematy blokowe itp. Teoria języków formalnych jest bardzo rozbudowana i nie

sposób wymienić i zdefiniować w krótkim skrypcie wszystkich ważnych pojęć, dlatego poniższa tabela przedstawia zestawienie dla najbardziej ogólnego podziału języków; osoby zainteresowane odsyłam do literatury ([8], [4]).

Typ	Gramatyki	Maszyny	Języki	Przykładowe produkcje
0	nieograniczone	maszyny Turinga	częściowo obliczalne	$a \to \varepsilon$ $ab \to a$ $ab \to ab$ $aAb \to A$ $A \to aB$
–	–	maszyny Turinga z własnością stopu	obliczalne	
1	kontekstowe	automaty ograniczone liniowo	kontekstowe	$A \to a$ $AB \to CB$ $aAb \to aBb$ $ABcd \to aCBcd$
2	bezkontekstowe	niedeterministyczne automaty skończone ze stosem	bezkontekstowe	$A \to \varepsilon$ $A \to a$ $A \to abc$ $A \to aBc$ $A \to BC$
3	regularne	automaty skończone	wyrażenia regularne	$A \to \varepsilon$ $A \to a$ $A \to abc$ $A \to aB$ $A \to abcB$

2.4.3 Problemy decyzyjne — porównanie

Języki regularne, automaty skończone

Języki regularne są bardzo prostymi językami (niewiele można opisać tak prostymi regułami jak wyrażenia regularne), ale większość problemów decyzyjnych jest dla nich rozstrzygalna, tzn. istnieją algorytmy, które zawsze potrafią zwrócić poprawną odpowiedź Tak lub Nie. Np. istnieje algorytm rozstrzygający, czy dany automat skończony akceptuje zero, skończoną liczbę, czy nieskończenie wiele słów. Algorytm taki można napisać korzystając z twierdzenia:

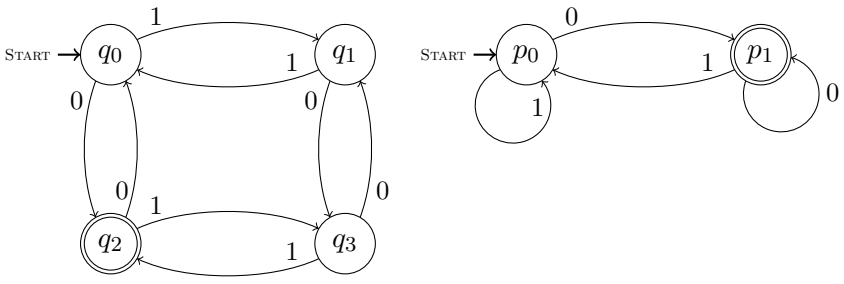
Twierdzenie 2.4.1. *Zbiór słów akceptowanych przez automat skończony M o n stanach jest:*

(1) niepusty *wtedy i tylko wtedy, gdy M akceptuje słowo o długości mniejszej od n;*

(2) nieskończony*, jeżeli M akceptuje słowo o długości l, gdzie $n \leqslant l < 2n$.*

Istnieje również algorytm rozstrzygania, czy dwa automaty skończone są równoważne (tzn. czy akceptują ten sam język).

Przykład 14. Rozważmy dwa automaty takie, jak na rysunku poniżej. Pytamy, czy automaty te są równoważne, tzn. czy akceptują ten sam język.



Automaty nie są równoważne, gdyż

	0	1
$\{q_0, p_0\}$	$\{q_2, p_1\}$	$\{q_1, p_0\}$
$\{q_2, p_1\}$	$\{q_0, p_1\}$	

Po przeczytaniu pierwszego zera oba automaty są w stanie końcowym: pierwszy automat w q_2, a drugi w p_1. Po przeczytaniu kolejnego zera pierwszy automat przechodzi do stanu q_0, który nie jest stanem końcowym, a drugi pozostaje w p_1. Zatem widać, że drugi automat akceptuje słowo 00, a pierwszy — nie. ◁

Gramatyki bezkontekstowe

Gramatyki bezkontekstowe są nieco bardziej skomplikowane niż wyrażenia regularne, a co za tym idzie, języki bezkontekstowe są szerszą klasą niż języki regularne. Niestety: więcej można zdefiniować, ale mniej można rozstrzygnąć. Np. istnieją algorytmy pozwalające stwierdzić:

- Czy język generowany przez gramatykę bezkontekstową jest pusty, skończony czy nieskończony?
- Czy dane słowo $x \in T^*$ należy do języka danej gramatyki bezkontekstowej $G = (N, T, P, S)$, czyli czy $x \in L(G)$?

Nie istnieją jednak algorytmy, które potrafiłyby rozstrzygnąć:

- Czy dwie gramatyki bezkontekstowe G_1, G_2 są równoważne, tzn. czy $L(G_1) = L(G_2)$?
- Czy dopełnienie języka L pewnej gramatyki bezkontekstowej G jest językiem bezkontekstowym, tzn. czy istnieje gramatyka bezkontekstowa generująca zbiór wszystkich słów, które nie należą do języka L?
- Czy dopełnienie języka pewnej gramatyki bezkontekstowej jest językiem skończonym?

Gramatyki kontekstowe

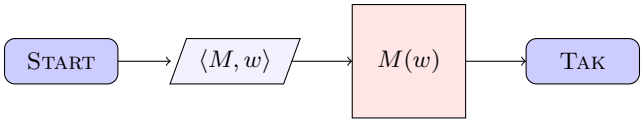
Istnieją algorytmy pozwalające stwierdzić, czy dane słowo należy do języka danej gramatyki kontekstowej.

Natomiast problem pustości języka generowanego przez gramatykę kontekstową jest nierozstrzygalny, tzn. nie istnieje algorytm, który dla zadanej gramatyki kontekstowej G stwierdza, czy $L(G) = \varnothing$.

Maszyny Turinga

Najbardziej znanym i podstawowym problemem, który jest *nierozstrzygalny* (tzn. nie istnieje algorytm, który zawsze potrafią zwrócić poprawną odpowiedź Tak lub Nie), ale jest *częściowo rozstrzygalny* (tzn. istnieje algorytm, które zawsze potrafi zwrócić poprawną odpowiedź Tak), jest *problem stopu* dla maszyn Turinga. Problem decyzyjny jest zdefiniowany w następujący:

Dane: maszyna Turinga M zakodowana w dowolny sposób oraz słowo w.
Problem: Czy maszyna M zatrzyma się na słowie wejściowym w?

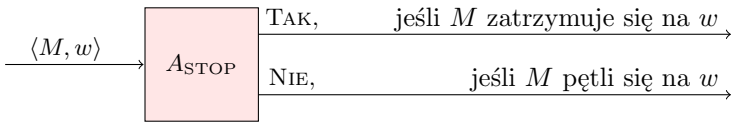


Rysunek 2.15: Algorytm odpowiadający Tak, jeśli M zatrzymuje się na w; stosujemy $\langle M, w\rangle$ jako symboliczny zapis kodu maszyny M i słowa w.

Algorytm odpowiadający Tak, jeśli M zatrzymuje się na w:

```
wczytaj kod maszyny M i słowo w;
uruchom M na słowie w i czekaj na wynik;
jeżeli M zatrzyma się na w, odpowiedz Tak.
```

Powyższy algorytm nie ma własności stopu: jeżeli M pętli się na słowie w, to nie otrzymamy żadnej odpowiedzi. Interesuje nas zatem pytanie, czy istnieje algorytm A_{STOP} z własnością stopu:

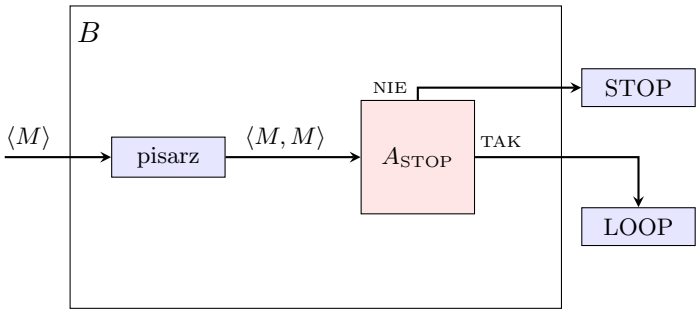


Rysunek 2.16: Hipotetyczna maszyna rozstrzygająca problem stopu.

Twierdzenie 2.4.2. *Nie istnieje uniwersalny algorytm rozstrzygający problem stopu dla wszystkich algorytmów.*

Dowód. Załóżmy, że taki algorytm A_{STOP} istnieje. Konstruujemy inny algorytm, który nazwiemy B:

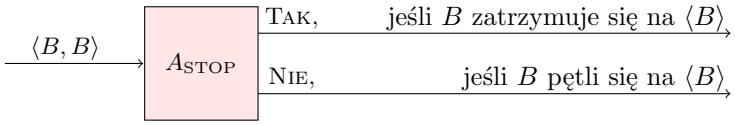
```
wczytaj kod maszyny M;
skopiuj M; // pisarz
uruchom A_STOP na ⟨M, M⟩;
jeżeli A_STOP(⟨M, M⟩) odpowie TAK, zapętlij program;
w przecinym przypadku, odpowiedz TAK.
```



Rysunek 2.17: Algorytm B

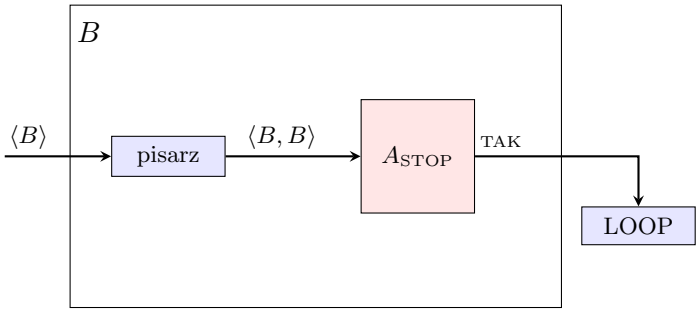
Algorytm B został napisany i można go teraz zakodować, czyli znaleźć kod $\langle B \rangle$. Kodowanie jest operacją czysto mechaniczną — robi to każdy kompilator. Prześledźmy działanie programu B na jakimś słowie. Wybierzmy słowo $\langle B \rangle$ — jest dobre jak każde inne.

Pamiętamy, że algorytm A_{STOP} zawsze udziela poprawnej odpowiedzi, zatem na pytanie, czy algorytm B zatrzymuje się na słowie $\langle B \rangle$, odpowie:



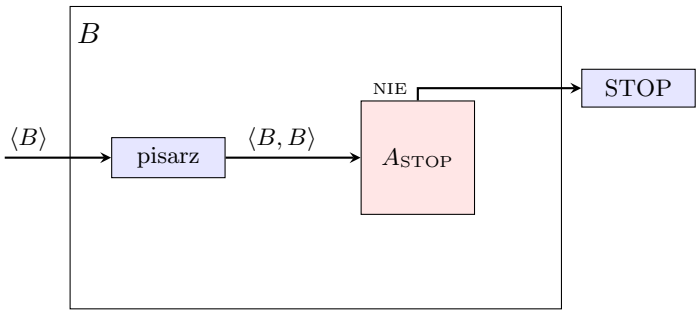
Ponieważ nie znamy słowa $\langle B \rangle$, to rozważmy oba przypadki;

1. Załóżmy, że A_{STOP} odpowiedział TAK, czyli że B zatrzymuje się na $\langle B \rangle$, ale jak widać na poniższym schemacie



maszyna B pętli się! Pętli się na słowie $\langle B \rangle$! Czyli sprzeczność — tak być nie może. Zatem A_{STOP} musiał udzielić innej odpowiedzi.

2. Załóżmy zatem, że A_{STOP} odpowiedział NIE, czyli że B pętli się na $\langle B \rangle$, ale jak widać



maszyna B zatrzymuje się na słowie $\langle B \rangle$! Czyli ponownie sprzeczność — tak też być nie może! □

Jak widać, założenie istnienia algorytmu A_{STOP} rozstrzygającego problem stopu doprowadziło nas do sprzeczności. Wniosek: problem jest stopu jest nierozstrzygalny.

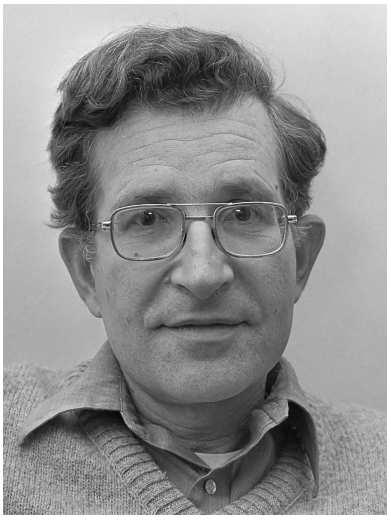
Podsumowanie

Istnienie algorytmu decyzyjnego zależy od klasy języka, a nie od zadanego pytania.

Poniższa tabelka ilustruje zależność istnienia algorytmów dla przykładowych problemów decyzyjnych od rozważanej klasy języków: pokazuje, że im bardziej skomplikowane są reguły opisu języków, tym trudniesze staje się algorytmiczne badanie ich własności. Zatem pytamy, czy następujące problemy są rozstrzygalne w poszczególnych klasach języków?

Typ	$x \in L(G)$	$L(G) = \varnothing$	$L(G_1) = L(G_2)$
0	NIE	NIE	NIE
1	TAK	NIE	NIE
2	TAK	TAK	NIE
3	TAK	TAK	TAK

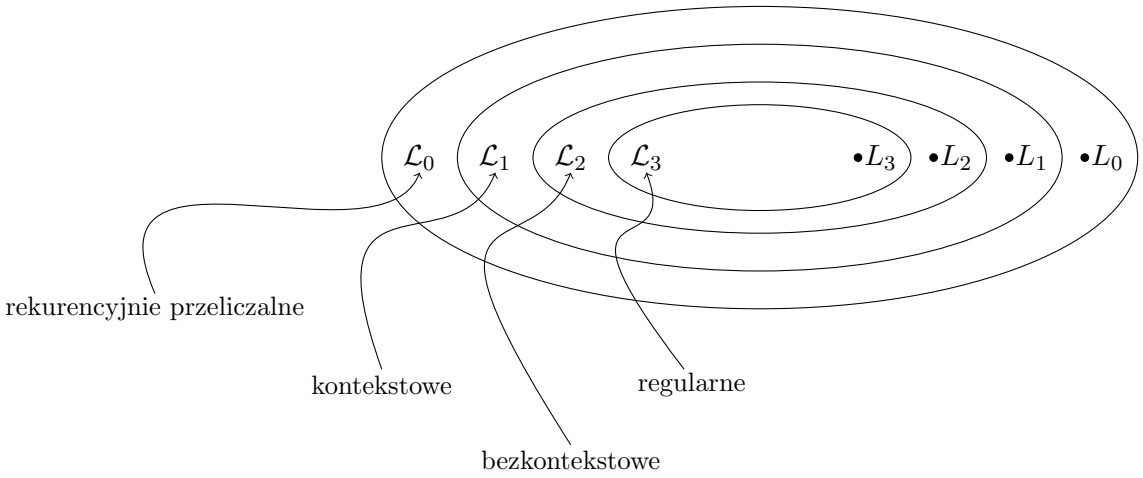
2.4.4 Hierarchia Chomsky'ego



Rysunek 2.18: Noam Chomsky[2]

Noam Chomsky zaproponował najbardziej znaną i powszechnie używaną klasyfikację języków formalnych. Jeżeli przez $\mathcal{L}_i$ oznaczymy zbiór wszystkich języków typu i, to

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0$$



Rysunek 2.19: Schemat zawierania się klas języków w hierarchii Chomsky'ego.

Przykładowe języki w poszczególnych klasach:

$$\begin{aligned}
\mathcal{L}_3 \ni L_3 &= \{a^n : n \in \mathbb{N}\},\\
\mathcal{L}_2 \ni L_2 &= \{a^n b^n : n \in \mathbb{N}\},\\
\mathcal{L}_1 \ni L_1 &= \{a^n b^n c^n : n \in \mathbb{N}\},\\
\mathcal{L}_0 \ni L_0 &= \{w \in \{0,1\}^* : w \text{ jest kodem TM, która zatrzymuje się na słowie pustym}\}.
\end{aligned}$$

Zauważmy, że na rysunku $L_0 \notin \mathcal{L}_1$, $L_1 \notin \mathcal{L}_2$, $L_2 \notin \mathcal{L}_3$ — każdy z tych faktów wymaga oddzielnego dowodu, ale obecnie są to już znane fakty. Na przykład dla pokazania, że $L_2 \notin \mathcal{L}_3$, wystarczy skorzystać z lematu o pompowaniu (zob. 2.3.1)

[2]Wikipedia, zdjęcie w domenie publicznej

Rozdział 3

Złożoność obliczeniowa

Na hierarchię Chomsky'ego, która odzwierciedla trudność problemów, można spojrzeć przez pryzmat ich złożoności obliczeniowej.

Nr	Gramatyki	Maszyny	Języki	Złożoność problemu „$x \in L$?"
0	nieograniczone	maszyny Turinga	rekurencyjnie przeliczalne	nierozstrzygalne
1	kontekstowe	maszyny Turinga ograniczone liniowo	kontekstowe	NSPACE(n)
2	bezkontekstowe	automaty skończone ze stosem	bezkontekstowe	P
3	regularne	automaty skończone	wyrażenia regularne	SPACE(1)

Dla wyjaśnienia skrótów NSPACE, P, SPACE konieczne jest zapoznanie się z klasyfikacją modeli obliczeń i podstawową terminologią związaną ze złożonością obliczeniową.

3.1 Złożoność — zużywane zasoby

Jednym z przełomowych niegdyś odkryć było stwierdzenie, że dwie dowolne maszyny, których hardware osiągnął dostateczny poziom złożoności i elastyczności, są sobie równoważne. Równoważność polega na tym, że dla dowolnych dwóch maszyn A i B można napisać takie programy, że dla dowolnych danych wejściowych maszyny te zwrócą ten sam wynik. Nie mówimy tu ani o potrzebnej ilości czasu, ani pamięci — te mogą się istotnie różnić.

Zatem aby stwierdzić, czy istnieje algorytm, wystarczy ograniczyć się do jednotaśmowej, deterministycznej maszyny Turinga, ponieważ we współczesnym świecie nie istnieje model obliczeń, który byłby od takiej maszyny „silniejszy". W rozdziale 2.4 widzieliśmy przykłady pytań, które miały odpowiedź lub nie, w zależności od przyjętego modelu. Zawsze jednak jeśli istniał algorytm, który rozwiązywał dany problem, to mógł być napisany na jednotaśmowej, deterministycznej maszynie Turinga (1-DTM). W teorii złożoności używa się różnych modeli obliczeń, gdyż w zależności od przyjętego modelu, problem lub algorytm mogą mieć różne własności.

Definicja 3.1.1. *Czas* $T(M, w)$ *obliczenia* maszyny Turinga M na konkretnym słowie wejściowym w to liczba kroków wykonanych, zanim maszyna się zatrzymała. Jeżeli M nigdy się nie zatrzyma, to czas obliczenia jest nieskończony, czyli $T(M, w) = \infty$.

Zdefiniowanie czasu obliczenia dla konkretnego wejścia jest jedynie początkiem. Tym, co nas naprawdę interesuje, jest czas potrzebny do rozwiązania dowolnej instancji problemu, gdzie *instancją* nazywamy konkretny zestaw danych z dziedziny problemu. Poniżej zakładamy, że czas obliczeń określamy jedynie dla maszyn z własnością stopu.

Definicja 3.1.2. Funkcję $f : \mathbb{N} \to \mathbb{R}^+$ nazywamy *złożonością czasową* maszyny Turinga M, jeżeli dla dowolnego $n \in \mathbb{N}$ *

$$f(n) = \max\{T(M, w) : |w| = n\}.$$

Zatem mówimy, że maszyna M działa w czasie $f(n)$, jeżeli największy czas obliczeń dla dowolnego słowa o długości n wynosi $f(n)$.

Definicja 3.1.3. *Pamięć* $S(M, w)$ potrzebna maszynie M do obliczeń na konkretnym słowie wejściowym w to liczba komórek taśmy, które zostały odwiedzone przez głowicę, zanim maszyna się zatrzymała. Jeżeli M nigdy się nie zatrzyma, to przyjmujemy, że pamięć potrzebna do obliczenia jest nieokreślona. *

Dlaczego chcemy wyróżnić sytuację, gdy maszyna się nie zatrzymuje? Wyobraźmy sobie maszynę, która pętli się na danym słowie korzystając tylko z jednej komórki taśmy. Nie chcemy „cieszyć się" niskim zużyciem pamięci przez maszynę, która nie nadaje się do faktycznych obliczeń.

Definicja 3.1.4. Funkcję $f : \mathbb{N} \to \mathbb{R}^+$ nazywamy *złożonością pamięciową* dla maszyny Turinga M z własnością stopu, jeżeli dla dowolnego $n \in \mathbb{N}$

$$f(n) = \max\{S(M, w) : w \in \Sigma\}.$$

3.2 Notacja $\mathcal{O}$ i o

Dokładne oszacowanie liczby kroków jakiegoś algorytmu dla dowolnych danych wejściowych w większości przypadków jest niezwykle trudne, a czasem wręcz niemożliwe. Zwykle zadowalamy się oszacowaniem pesymistycznej liczby wykonanych kroków, czyli liczby kroków wykonanych w najgorszym możliwym przypadku. Narzędziem dla porównywania zachowania oszacowań przy coraz większym rozmiarze danych wejściowych jest notacja asymptotycznego tempa wzrostu:

- $f(n) = \mathcal{O}(g(n))$ — f jest co najwyżej rzędu g,
- $f(n) = o(g(n))$ — f jest niższego rzędu niż g,
- $f(n) = \Omega(g(n))$ — f jest co najmniej rzędu g, *
- $f(n) = \omega(g(n))$ — f jest wyższego rzędu niż g,
- $f(n) = \Theta(g(n))$ — f jest dokładnie rzędu g.

Bardziej formalnie, np. $f(n) = \mathcal{O}(g(n))$ oznacza, że od pewnego miejsca począwszy funkcja $f(n)$ jest od góry ograniczana przez $cg(n)$, gdzie c jest dowolną stałą. W zapisie tym nadużywamy znaku równości; w gruncie rzeczy lepszy byłby znak $\in$, tradycyjnie piszemy jednak $=$.

Definicja 3.2.1 ($\mathcal{O}$ duże)**.** Niech $f, g : \mathbb{N} \to \mathbb{R}^+$. Mówimy, że $g(n)$ jest *asymptotycznym górnym ograniczeniem* $f(n)$ i zapisujemy $f(n) = \mathcal{O}(g(n))$, jeżeli

$$\exists c > 0, n_0 > 0 \ \forall n \geqslant n_0 : \ f(n) \leqslant cg(n).$$

Przykład 15.

1. $2n = \mathcal{O}(n)$
2. $n \log n = \mathcal{O}(n^2)$
3. $n^2 \neq \mathcal{O}(n)$
4. $n^2 \neq \mathcal{O}(n \log^2 n)$
5. $3^n \neq \mathcal{O}(2^n)$ ◁

Definicja 3.2.2 (o małe)**.** Niech $f, g : \mathbb{N} \to \mathbb{R}^+$. Mówimy, że $f(n) = o(g(n))$, jeżeli

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Pozostałe symbole można zdefiniować za pomocą $\mathcal{O}$ i o:

- $f(n) = \Omega(g(n))$, jeżeli $g(n) = \mathcal{O}(f(n))$,
- $f(n) = \omega(g(n))$, jeżeli $g(n) = o(f(n))$,
- $f(n) = \Theta(g(n))$, jeżeli $f(n) = \mathcal{O}(g(n))$ i $g(n) = \mathcal{O}(f(n))$.

3.3 Modele obliczeń

Przykład 16. Chcemy napisać algorytm, rozstrzygający, czy słowo wejściowe jest postaci

$$\underbrace{0\ldots0}_{k}\underbrace{1\ldots1}_{k},$$

gdzie $k \geqslant 0$. Najlepszy algorytm jednotaśmowej maszyny Turinga rozstrzygający ten język:

- `Przejrzyj całą taśmę i` *odrzuć*`, jeśli na prawo od jedynek znajduje się jakieś zero.`
- `Dopóki na taśmie są zera i jedynki, powtarzaj:`
  - `Przejdź całą taśmę, sprawdzając, czy sumaryczna liczba zer i jedynek na taśmie jest parzysta, czy nieparzysta. Jeśli jest nieparzysta, to` *odrzuć.*
  - `Przejdź ponownie całą taśmę, wykreślając co drugie zero, począwszy od pierwszego, a następnie, wykreślając co drugą jedynkę, począwszy od pierwszej.`
- `Jeśli na taśmie nie pozostało ani jedno zero i ani jedna jedynka, to` *zaakceptuj.* `W przeciwnym przypadku` *odrzuć.*

Prześledźmy algorytm *krok po kroku* na przykładzie słowa $0^8 1^4$:

0_1	0_2	0_3	0_4	0_5	0_6	0_7	0_8	1_9	1_{10}	1_{11}	1_{12}
0_1	0_2	0_3	0_4	0_5	0_6	0_7	0_8	1_9	1_{10}	1_{11}	1_{12}
	0_1		0_2		0_3		0_4		1_5		1_6
	0_1		0_2		0_3		0_4		1_5		1_6
			0_1				0_2				1_3

Odpowiedź: słowo nie jest postaci $0^k 1^k$.
Ile operacji wykonuje algorytm na słowie wejściowym o długości n?

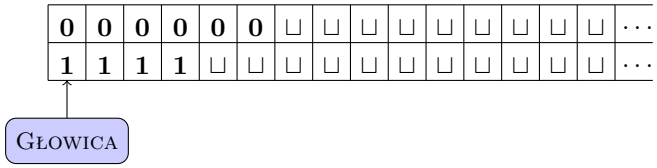
• `Przejrzyj całą taśmę i` *odrzuć*`, jeśli na prawo od jedynek znajduje się jakieś zero.`	1 raz
• `Dopóki na taśmie są zera i jedynki, powtarzaj: ...`	$1 + \log n$ razy
• `Jeśli na taśmie nie pozostało ani jedno zero i ani jedna jedynka, to` *zaakceptuj.* `W przeciwnym przypadku` *odrzuć.*	1 raz

To daje całościowy czas rzędu

$$\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n).$$

Jeśli maszyna ma drugą taśmę, to możemy napisać algorytm, który działa w czasie liniowym $\mathcal{O}(n)$. Algorytm działa tak, że po początkowym sprawdzeniu, czy na prawo od jedynek nie ma

zer, kopiuje jedynki na taśmę roboczą. Potem wykreśla równolegle zero z taśmy roboczej i jedynkę z wejścia.



◁

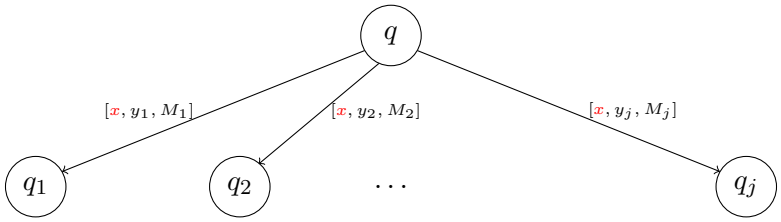
Przykładowe modele obliczeń równoważne obliczeniowo jednotaśmowej, deterministycznej maszynie Turinga:

- wielotaśmowa maszyna Turinga,
- niedeterministyczna maszyna Turinga, *
- układy logiczne,
- maszyna RAM,
- funkcje rekurencyjne.

Dodajmy, że modele równoważne obliczeniowo (mogące wykonać te same obliczenia co maszyna Turinga) mogą działać z zupełnie inną złożonością.

Niedeterministyczna maszyna Turinga (NTM)

W deterministycznej maszynie Turinga dla stanu q i symbolu x mamy jednoznacznie określoną akcję: nowy symbol, nowy stan i ruch głowicy. Natomiast w *niedeterministycznej maszynie Turinga* N * dla stanu q i symbolu x mamy zbiór możliwych akcji $\{(q_i, y_i, M_i)\colon i = 1, \ldots, k_{q,x},\ M_i \in \{L, R, -\}\}$. Liczbę $k = \max_{q,x} k_{q,x}$ nazywamy *stopniem niedeterminizmu* maszyny N. Na poniższym rysunku, ze względu na czytelność oznaczeń zamiast $k_{q,x}$ piszemy j. *



Definicja 3.3.1 (Akceptacja słowa)**.** Mówimy, że maszyna niedeterministyczna

- *akceptuje* słowo, jeżeli **istnieje** co najmniej jedna akceptująca ścieżka obliczeń,
- *odrzuca* słowo, jeżeli **żadna** ścieżka obliczeń nie prowadzi do stanu akceptującego.

Definicja 3.3.2 (Akceptacja języka)**.** Mówimy, że maszyna niedeterministyczna N *rozstrzyga* język L w czasie $f(n)$, jeżeli

- dla każdego słowa wejściowego x maszyna N decyduje, czy należy ono do języka L, czy nie;
- dla żadnego słowa wejściowego o długości n maszyna N nie ma ścieżek obliczeń dłuższych niż $f(n)$. *

Zauważmy, że $f(n)$ stanowi tu ograniczenie górne, które nie musi być realizowane.

3.4 Klasy złożoności

Niech n będzie długością słowa wejściowego. Rozważamy funkcje $f\colon \mathbb{N} \to \mathbb{R}^+$ takie, że $f(n) \geqslant \log n$.

Definicja 3.4.1.

- *Klasa złożoności czasowej* TIME($f(n)$) to zbiór wszystkich języków rozstrzyganych w czasie $\mathcal{O}(f(n))$ przez pewną DTM.
- *Klasa złożoności pamięciowej* SPACE($f(n)$) to zbiór wszystkich języków rozstrzyganych w pamięci $\mathcal{O}(f(n))$ przez pewną DTM.
- *Klasa złożoności czasowej* NTIME($f(n)$) to zbiór wszystkich języków rozstrzyganych w czasie $\mathcal{O}(f(n))$ przez pewną NDTM.
- *Klasa złożoności pamięciowej* NSPACE($f(n)$) to zbiór wszystkich języków rozstrzyganych w pamięci $\mathcal{O}(f(n))$ przez pewną NDTM. *

Fakt, że istnieje wiele funkcji, które nie są obliczane przez żadną maszynę Turinga, skłania nas do postawienia pytania: Czy wystarczy ograniczyć się do funkcji obliczalnych f, czyli funkcji obliczanych przez pewną maszynę Turinga? Okazuje się, że tak. Ale okazuje się również, że funkcje * obliczalne to zdecydowanie za dużo. Jeżeli nie przyjmiemy dodatkowych ograniczeń, otrzymamy całkowicie nieintuicyjne rezultaty:

*

Twierdzenie 3.4.1 (o luce)**.** *Istnieje funkcja $f : \mathbb{N} \to \mathbb{N}$ obliczana przez maszynę Turinga z własnością stopu taka, że*

$$\mathrm{TIME}(f(n)) = \mathrm{TIME}(2^{f(n)}).$$

Dowód. Zob. [5]. □

Aby zabezpieczyć się przed nieintuicyjnymi klasami złożoności, stawiamy ograniczenia na funkcję $f(n)$. *Maszyna Turinga z wejściem i wyjściem*, o której mowa w poniższej definicji, to jeszcze jeden wariant wielotaśmowej maszyny deterministycznej, która posiada dwie wyróżnione taśmy: wejściową i wyjściową (oraz taśmy robocze). Taśma wejściowa może być jedynie odczytywana. Taśma wyjściowa służy tylko do zapisu, a głowica może się po niej poruszać tylko w prawo.

Definicja 3.4.2. Funkcja $f : \mathbb{N} \to \mathbb{N}$ jest *właściwą funkcją złożoności,* jeżeli:

- jest funkcją niemalejącą,
- istnieje wielotaśmowa maszyna Turinga *z wejściem i wyjściem,* która mając na wejściu x * oblicza ciąg jedynek o długości $f(|x|)$ zatrzymując się po $\mathcal{O}(|x| + f(|x|))$ krokach i zużywając $\mathcal{O}(f(|x|))$ pamięci (oprócz wejścia i wyjścia).

Czasem zamiast konkretnej funkcji f w definicji klasy złożoności używamy rodziny funkcji sparametryzowanej przez $j > 0$. Zapis postaci n^k jest tradycyjnie (nad)używany na oznaczenie sumy po wszystkich wykładnikach dodatnich. Stosujemy następujące oznaczenia: *

- $\mathrm{P} = \mathrm{TIME}(n^k) = \bigcup_{j>0} \mathrm{TIME}(n^j)$,
- $\mathrm{NP} = \mathrm{NTIME}(n^k) = \bigcup_{j>0} \mathrm{NTIME}(n^j)$,
- $\mathrm{EXP} = \mathrm{TIME}(2^{n^k})$,
- $\mathrm{PSPACE} = \mathrm{SPACE}(n^k)$,
- $\mathrm{NPSPACE} = \mathrm{NSPACE}(n^k)$,
- $\mathrm{L} = \mathrm{SPACE}(\log n)$,
- $\mathrm{NL} = \mathrm{NSPACE}(\log n)$.

Twierdzenie 3.4.2. *

1. $\mathrm{TIME}(f(n)) \subseteq \mathrm{NTIME}(f(n))$,
2. $\mathrm{SPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$.

Dowód. Dowód jest oczywisty, gdyż każda deterministyczna maszyna Turinga jest maszyną niedeterministyczną ze stopniem niedeterminizmu równym 1. □

Słynne pytanie za \$1000000 stanowiące jeden z problemów milenijnych Instytutu Matematycznego Claya[a]:

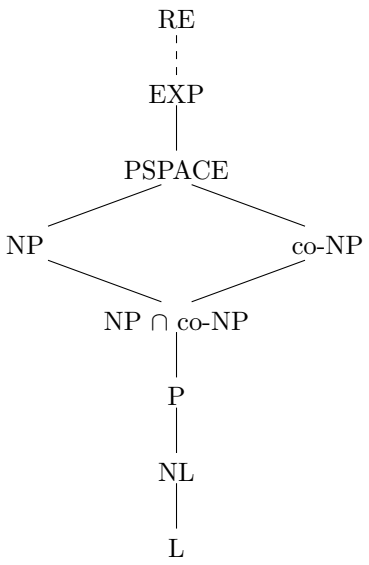
$$\text{Czy P} = \text{NP?}$$

[a] https://www.claymath.org/millennium/p-vs-np/

3.5 Hierarchia klas złożoności

Na poniższym diagramie Hassego przez

- RE oznaczamy wszystkie języki obliczalne przez dowolną maszynę Turinga (nawet bez własności stopu),
- co-NP klasę złożoności: problem decyzyjny należy do klasy co-NP wtedy i tylko wtedy, gdy jego dopełnienie należy do klasy NP. *



Z powyższego diagramu możemy odczytać, że

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}\,.$$

Wiadomo, że $\text{L} \neq \text{PSPACE}$, zatem przynajmniej jedna z powyższych inkluzji jest ścisła, ale nie wiadomo, która!

3.6 Koszt algorytmu a złożoność problemu

Złożoność (pesymistyczna) algorytmu to funkcja rozmiaru danych podająca maksymalne zużycia ustalonego zasobu (zwykle czas lub pamięć) przez algorytm dla danych rozmiaru n (zob. definicje 3.1.2 i 3.1.4).

Złożoność problemu to minimum ze złożoności wszystkich algorytmów rozwiązujących dany problem.

Dla rozróżnienia tych dwóch pojęć złożoność algorytmu zwykle nazywamy kosztem.

3.6.1 Proste algorytmy sortowania

Załóżmy, że dane (liczby) do sortowania przechowujemy w tablicy A. Sortowanie polega na permutowaniu tych liczb, aż do chwili osiągnięcia uporządkowania

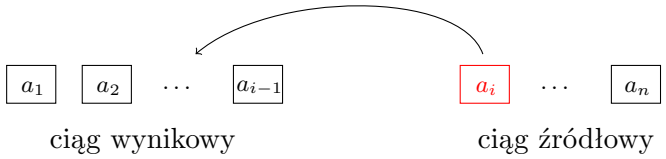
$$A[1] \leqslant A[2] \leqslant \ldots \leqslant A[n].$$

Proste metody sortowania można podzielić na trzy grupy:

- sortowanie przez wstawianie,
- sortowanie przez wybieranie,
- sortowanie przez zamianę.

Sortowanie przez proste wstawianie

Obiekty są podzielone umownie na ciąg wynikowy $a_1, \ldots, a_{i-1}$ i ciąg źródłowy $a_i, \ldots, a_n$. W każdym kroku pierwszy, tzn. skrajnie lewy element ciągu źródłowego a_i przenosi się do ciągu wynikowego wstawiając go w odpowiednim miejscu. Proces rozpoczyna się od $i = 2$, czyli od jednoelementowego ciągu wynikowego i w każdym kroku i zwiększa się o jeden.



Rysunek 3.1: Sortowanie przez proste wstawianie

Przykład 17. *

ciąg początkowy	5	8	2	1	9	3	6
	[5]	[8]	2	1	9	3	6
	[5	8]	[2]	1	9	3	6
	[2	5	8]	[1]	9	3	6
	[1	2	5	8]	[9]	3	6
	[1	2	5	8	9]	[3]	6
	[1	2	3	5	8	9]	[6]
ciąg posortowany	1	2	3	5	6	8	9

◁

Algorytm prostego wstawiania przedstawiamy poniżej. Zauważmy, że sortowane liczby przechowujemy w tablicy, numerując je od 1. Element $A[0]$ nie jest zajęty i w algorytmie wykorzystujemy go w roli tzw. wartownika, gwarantującego poprawne zakończenie pętli w każdym przypadku. *

```
void prosteWstawianie(int A[])
{
   int j, x;

   for (int i = 2; i <= n; i++) {
      x = A[i];
      A[0] = x;
      j = i - 1;
      while (x < A[j]) {
         A[j+1] = A[j];
```

```
            j = j-1;
        }
        A[j+1] = x;
    }
}
```

*

Uwaga ogólna do liczenia kosztu algorytmu
Powszechnie przyjętą praktyką jest zliczanie operacji dominujących. Ponieważ operacji na indeksach jest (z dokładnością do rzędu wielkości) nie więcej niż operacji na elementach tablicy (które mogą być bardziej kosztowne ze względu na większy rozmiar), zliczamy tylko te drugie.

Analiza pesymistycznego kosztu prostego wstawiania:

- kroków algorytmu jest $n-1$,
- P_{o_i} — liczba porównań w i-tym kroku wynosi co najwyżej i, *
- P_{r_i} — liczba podstawień elementów tablicy (razem z ustawieniem „wartownika") wynosi $P_{o_i}+2$.

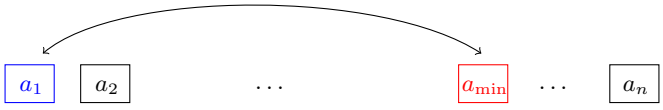
Wynika stąd, że pesymistycznie *

$$P_o = \sum_{i=2}^{n} i = \frac{2+n}{2}(n-1) = \frac{1}{2}(n^2+n-2),$$
$$P_r = \sum_{i=2}^{n} (i+2) = \frac{1}{2}(n^2+5n-6).$$

Jak widać, złożoność algorytmu prostego wstawiania jest $\mathcal{O}(n^2)$, chociaż dla ciągu pierwotnie już posortowanego algorytm działa w czasie n.

Sortowanie przez proste wybieranie

Wybieramy najmniejszy element spośród $a_1, \ldots, a_n$ i wymieniamy go z elementem a_1. Następnie wybieramy najmniejszy element spośród $a_2, \ldots, a_n$ i wymieniamy go z elementem a_2. Proces powtarzamy, aż pozostanie jeden element — największy.



Rysunek 3.2: Sortowanie przez proste wybieranie

Przykład 18.

ciąg początkowy	5	8	2	1	9	3	6
	[5]	8	2	[1]	9	3	6
	1	[8]	[2]	5	9	3	6
	1	2	[8]	5	9	[3]	6
	1	2	3	[5]	9	8	6
	1	2	3	5	[9]	8	[6]
	1	2	3	5	6	[8]	9
ciąg posortowany	1	2	3	5	6	8	9

Algorytm prostego wybierania:

```
void prosteWybieranie(int A[])
{
   int k, x;

   for (int i = 1; i <= n-1; i++) {
      k = i;  x = A[i];
      for (int j = i + 1; j <= n; j++) {
         if (A[j] < x) {
            k = j; x = A[j];
         }
      A[k] = A[i]; A[i] = x;
      }
   }
}
```

Analiza pesymistycznego kosztu algorytmu prostego wybierania:

- P_{o_i} — liczba porównań w i-tym kroku wynosi dokładnie $n - i$,
- P_{r_i} — liczba podstawień w i-tym kroku wynosi $P_{o_i} + 3$.

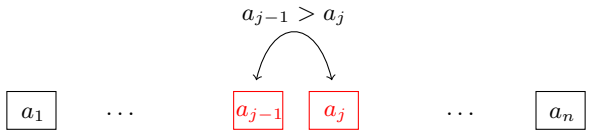
Wynika stąd, że

$$P_o = \sum_{i=2}^{n}(n-i) = \frac{1}{2}(n^2 - 3n + 2),$$
$$P_r = \sum_{i=2}^{n}((n-i)+3) = \frac{1}{2}(n^2 - 3n + 2) + 3(n-1).$$

Jak widać, ponownie pesymistyczna złożoność algorytmu prostego wybierania jest $\mathcal{O}(n^2)$.

Sortowanie przez prostą zamianę

Porównujemy element a_j z a_{j-1} i jeżeli są w niewłaściwej kolejności, to zamieniamy je. W każdej pętli element najmniejszy jest „wypychany" na początek.



Rysunek 3.3: Sortowanie przez prostą zamianę

Przykład 19.

ciąg początkowy	5	8	2	1	9	3	6
	5	8	2	1	9	**3**	**6**
	5	8	2	1	**9**	**3**	6
	5	8	2	**1**	**3**	9	6
	5	8	**2**	**1**	3	9	6
	5	**8**	**1**	2	3	9	6
	5	**1**	8	2	3	9	6
	1	5	8	2	3	**9**	**6**
	1	5	8	2	**3**	**6**	9
	1	5	8	**2**	**3**	6	9

...

Algorytm sortowania bąbelkowego (typowy wariant sortowania przez prostą zamianę):

```
void sortowanieBabelkowe(int A[])
{
   int x;

   for (int i = 2; i <= n; i++) {
      for (int j = n; j >= i; j--) {
         if (A[j-1] > A[j]) {
            x = A[j-1];
            A[j-1] = A[j];
            A[j] = x;
         }
      }
   }
}
```

Analiza kosztu algorytmu sortowania bąbelkowego:

- liczba porównań zawsze (niezależnie od danych wejściowych) wynosi $P_o = \frac{1}{2}(n^2 - 3n + 2) + (n - 1)$,
- a liczba przesunięć (w przypadku pesymistycznym) to $P_r = 3 \cdot P_o$.

Ponownie złożoność algorytmu jest $\mathcal{O}(n^2)$.

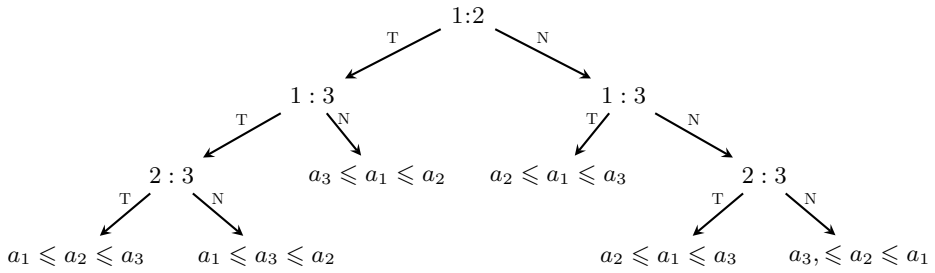
3.6.2 Dolne ograniczenie złożoności czasowej sortowania

Rozważmy, jaka jest dolna granica złożoności czasowej algorytmów dla problemu sortowania przez porównania.

Sortowanie przez porównania to takie sortowanie, w którym końcowa permutacja sortowanych elementów jest osiągana wyłącznie na podstawie porównań elementów. Nie każde sortowanie jest sortowaniem takiego typu. Są algorytmy, które np. „zaglądają" do maszynowej reprezentacji elementów.

Dla takiego rodzaju sortowania można skonstruować *drzewo decyzyjne*, które jest drzewem binarnym: w węzłach drzewa występują wszystkie możliwe pary indeksów i, j, $1 \leqslant i, j \leqslant n$, a krawędzie są etykietowane przez `Tak` lub `Nie`. W liściach drzewa są wszystkie możliwe permutacje elementów sortowanego ciągu.

Przykład 20. Przykład drzewa decyzyjnego dla trzech elementów



Drzewo decyzyjne dla n elementów ma $n!$ liści, a jego wysokość (liczba krawędzi na najdłuższej ścieżce) wynosi $\lceil \log_2 n! \rceil$.

Niech $P_{\min}(n)$ oznacza minimalną liczbę porównań wystarczającą do posortowania n elementów. Jest to wysokość drzewa decyzyjnego:

$$P_{\min}(n) = \lceil \log n! \rceil \geqslant n \log n - 1.45n.$$

Zatem algorytm sortujący przez porównania nie może być w ogólnym przypadku szybszy niż $n \log n$ (w szczególnym przypadku może, np. jeżeli do algorytmu sortowania bąbelkowego podamy ciąg posortowany, to wykona zaledwie n porównań i zakończy działanie).

Wniosek:

Złożoność problemu sortowania wynosi $\mathcal{O}(n \log n)$.

Rozdział 4

Reprezentacja maszynowa liczb

Sposób zapisu liczb wewnątrz komputera wydaje się rzeczą oczywistą, tymczasem i tu można napotkać sporo niespodzianek. Często biorą się one z nieuwzględnienia niuansów semantyki języka programowania (np. nieoczywistych konwersji typów) lub z elementarnego przeoczenia, że liczby przetwarzane w komputerze to tylko pewien skończony podzbiór. W każdej konkretnej sytuacji warto zatem wiedzieć przynajmniej to, jakim podzbiorem dysponujemy. Niniejszy rozdział omawia podstawowe zasady reprezentacji liczb, co pozwala zwrócić uwagę na te przypadłości.

Rozdział ten jest jedynym, który zawiera bezpośrednie odniesienia do rzeczywistej, fizycznej konstrukcji komputerów. Jest to konsekwencja tego, że mówimy o problemach leżących na styku sprzętu i oprogramowania. Z podobnego powodu w wielu miejscach odwołujemy się do przykładów z języka C, który jest najbardziej „maszynowym" z popularnych języków programowania.

Warto uświadomić sobie, że często mówimy o *liczbach*, w istocie mając na myśli ich *reprezentacje*. Frazy takie jak „liczba dziesiętna" lub „niebieska liczba u góry rysunku" odnoszą się przecież nie do abstrakcyjnych liczb, lecz do ich reprezentacji. Żeby przetwarzać liczby (i jakiekolwiek inne dane) — czy to we własnym umyśle, czy w komputerze — potrzebujemy reprezentacji. W przypadku komputerów niewątpliwie potrzebujemy reprezentacji skończonych, a najlepszą wydajność jesteśmy w stanie osiągnąć, gdy rozmiar reprezentacji ustalimy z góry i to na umiarkowanym poziomie. Rozmaite typy liczbowe w językach programowania odpowiadają różnym kombinacjom rodzaju reprezentacji i jej rozmiaru, ale w żadnym przypadku nie są to prawdziwe liczby całkowite lub rzeczywiste. Innymi słowy, `int` $\neq \mathbb{N}$ oraz `float` $\neq \mathbb{R}$.

4.1 Liczby całkowite

4.1.1 Zagadki C-owe[1]

Zanim przejdziemy do ścisłego omówienia reprezentacji liczb całkowitych, przedstawiamy kilka zagadek dla miłośników języka C. Choć powstał on pół wieku temu — na początku lat 70. XX wieku — wciąż odgrywa dużą rolę jako punkt odniesienia dla wielu innych popularnych języków; jest też najbardziej oczywistym wyborem, gdy chcemy możliwie bezpośrednio wykonywać operacje na danych w pamięci.

Zakładamy, że maszyna jest 64-bitowa i reprezentuje liczby całkowite za pomocą reprezentacji uzupełnieniowej do dwóch. Przy założeniu wskazanych w ramce deklaracji, dla każdego podanego wyrażenia należy albo uzasadnić, że jest prawdziwe dla dowolnych wartości `x` i `y`, albo podać przykład, dla którego jest nieprawdziwe.

[1] Pierwszeństwo w popularyzacji przymiotnika „C-owy" należy do autora cenionych i niezwykle popularnych w latach 80. i 90. książek o językach programowania, Jana Bieleckiego. Żmudna kwerenda lingwistyczna prowadzi do wniosku, że czytelnicy zazwyczaj wymawiali ów przymiotnik jako „ceowy".

```
int x = ...;   // tu podstawiamy dowolną wartość
int y = ...;   // jw.
unsigned ux = x;
```

```
      x < 0  ⇒  (x * 2) < 0
    ux >= 0
x & 7 == 7  ⇒  (x << 62) < 0
    ux > -1
      x > y  ⇒  -x < -y
  x * x >= 0
x > 0 && y > 0  ⇒  x + y > 0
      x >= 0  ⇒  -x <= 0
      x <= 0  ⇒  -x >= 0
```

4.1.2 Kodowanie liczb całkowitych

Będziemy się zajmowali dwoma najczęściej wykorzystywanymi w komputerach kodowaniami liczb: najprostszą *reprezentacją binarną bez znaku* oraz *reprezentacją uzupełnieniową do dwóch* (którą będziemy skrótowo oznaczali ud2). W popularnych językach programowania ta pierwsza stosowana jest do wartości (zmiennych, wyrażeń, funkcji), których typ ma w nazwie słowo unsigned; ta druga — do wszelkich pozostałych wartości całkowitych, czyli do najpowszechniejszych w programowaniu wartości typu int/integer (z ewentualnymi modyfikatorami).

W poniższym zapisie funkcje B2U i B2T oznaczają interpretację ciągu bitów X, gdy przyjmiemy, że reprezentuje on liczbę bez znaku (B2U, *Bits to Unsigned*) lub liczbę ze znakiem zapisaną jako ud2 (B2T, *Bits to Two's complement*). Funkcje te stanowią więc definicje omawianych reprezentacji. Stała w oznacza liczbę bitów, których używamy do reprezentowania liczby; potocznie nazywa się tę wielkość rozmiarem zmiennej lub długością (lub szerokością) słowa. Najczęściej spotykane wartości w to 8, 16, 32, 64 i ewentualnie następne potęgi dwójki. Warto zwrócić uwagę na to, że choć każdy komputer ma pewną naturalną długość słowa, wynikającą z jego fizycznej konstrukcji, to różne języki programowania, a niejednokrotnie nawet różne realizacje tego samego języka stosują reprezentacje o różnej długości.

- bez znaku:

$$\mathrm{B2U}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i.$$

Czynniki x_i oznaczają tu kolejne bity ciagu X, zaś czynniki 2^i zwykle zwane są wagami. Bity numerujemy tak, że najbardziej znaczący („najbardziej lewy") bit ciągu to x_{w-1}, a najmniej znaczący („najbardziej prawy") to x_0. Jest to prosta, klasyczna notacja, znana każdemu od urodzenia: *

$$\begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & = & (33)_{10}. \\ 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

Indeksu dolnego „$_{10}$" używamy dla zaznaczenia, że liczba zapisana jest w reprezentacji dziesiętnej. Analogicznie, w razie potrzeby będziemy również używali indeksu „$_2$".

Uwaga: powyższego sposobu numerowania i nazywania bitów w ciągu będziemy konsekwentnie używali w całym tekście. Odstępstwa od tej zasady będą wskazane expressis verbis.

- ud2 (ze znakiem):

$$\mathrm{B2T}(X) = -\underbrace{x_{w-1}}_{\text{bit znaku}} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i.$$

W ud2 najbardziej znaczący bit wskazuje na znak liczby:

– liczby nieujemne mają w tym miejscu 0,
– liczby ujemne mają 1, np.

$$\begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & = & (-31)_{10} \\ 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

Zauważmy, że różnica między reprezentacją bez znaku a reprezentacją ud2 polega jedynie na interpretacji wartości najbardziej znaczącego bitu — jako dodatniej lub ujemnej.

Przykład 21. Kolejny, bardziej szczegółowy przykład, jak reprezentowane są liczby dodatnie i ujemne. Załóżmy, że mamy deklarację w C:

```
short int x = 12345;
short int y = -12345;
```

Załóżmy, że typ `short int` to dwa bajty, czyli 16 bitów. Wówczas (odstępy w kolumnie „ud2" służą jedynie zwiększeniu czytelności zapisu):

	dziesiętnie	ud2
x	12345	00110000 00111001
y	-12345	11001111 11000111

Dlaczego? Przeliczenie obrazuje poniższa tabelka. Szarą czcionką zaznaczono kolejne bity reprezentacji ud2 dwóch przykładowych liczb, czarną — odpowiadające im wartości z uwzględnieniem wag poszczególnych bitów. Suma w dolnym wierszu to oczywiście interpretacja ciągu bitów w reprezentacji ud2.

Wagi	x		y	
1	1	1	1	1
2	0	0	1	2
4	0	0	1	4
8	1	8	0	0
16	1	16	0	0
32	1	32	0	0
64	0	0	1	64
128	0	0	1	128
256	0	0	1	256
512	0	0	1	512
1024	0	0	1	1024
2048	0	0	1	2048
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
−32768	0	0	1	−32768
Suma		12345		−12345

◁

4.1.3 Zakresy wartości

Szczególny sposób wykorzystania jednego bitu w reprezentacji ud2 powoduje brak symetrii zakresu reprezentacji liczb dodatnich i ujemnych; po stronie ujemnej mamy o jedną liczbę więcej. Zobaczmy, jak wyglądają reprezentacje najmniejszej i największe możliwej do reprezentowania liczby przy danej długości słowa w.

- Reprezentacja bez znaku:
  $U_{\min} = 0$, czyli $000\ldots0$,
  $U_{\max} = 2^w - 1$, czyli $111\ldots1$.
- Reprezentacja ud2:
  $T_{\min} = -2^{w-1}$, czyli $100\ldots0$,
  $T_{\max} = 2^{w-1} - 1$, czyli $011\ldots1$,
  szczególną postać ma liczba -1, czyli $111\ldots1$.
- Obserwacje:
  $|T_{\min}| = T_{\max} + 1$, zatem zakres jest niesymetryczny,
  $U_{\max} = 2T_{\max} + 1$.

Przykład 22.

- Zobaczmy, jakie są skrajne i pewne wyróżnione wartości dla słów o długości 16 bitów ($w = 16$):

	dziesiętnie	ud2
$U_{\max}$	65535	11111111 11111111
$T_{\max}$	32767	01111111 11111111
$T_{\min}$	-32768	10000000 00000000
-1	-1	11111111 11111111
0	0	00000000 00000000

- Wartości dla różnych rozmiarów słów:

w:	8	16	32	64
$U_{\max}$	255	65 535	4 294 967 295	18 446 744 073 709 551 615
$T_{\max}$	127	32 767	2 147 483 647	9 223 372 036 854 775 807
$T_{\min}$	-128	-32 768	-2 147 483 648	- 9 223 372 036 854 775 808

4.1.4 Liczby całkowite w języku C — parę uwag

Zmienne typu `int` domyślnie (tzn. jeśli nie napiszemy `unsigned`) są liczbami ze znakiem. Podobną właściwość ma typ `char`, służący do przechowywania znaków. Często myślimy, że jeżeli podstawimy zmienną `char` pod zmienną `int`, to otrzymamy wartość z przedziału od 0 do 255, a tymczasem zwykle będzie to wartość od -128 do 127.

Jeżeli chcemy napisać przenośny kod, najlepiej sprawdzić rozmiary i domyślny sposób reprezentacji poszczególnych typów, sprawdzając dokumentację lub wyliczając `sizeof(...)`; rozmiar podawany jest jako wielokrotność rozmiaru typu `char`, który zwykle jest jednobajtowy.

Kolejnym drażliwym tematem jest reprezentacja znaków. Powszechny standard Unicode przewiduje kody dla setek tysięcy możliwych znaków i w praktyce najczęściej stosowane jest kodowanie UTF-8, w którym znaki są reprezentowane za pomocą ciągów o różnej długości, od 1 do 4 bajtów, przy czym tylko podstawowe znaki mają kody jednobajtowe. W konsekwencji znika prosta odpowiedniość między znakami a (małymi) liczbami całkowitymi.

Charakterystyczne dla danej platformy (tj. maszyny i wersji języka) wartości wyznaczające zakresy liczb można znaleźć np. w pliku `limits.h`. Są tam deklaracje takich stałych, jak `ULONG_MAX`, `LONG_MAX`, `LONG_MIN`, określające minimalne i maksymalne wartości w poszczególnych typach.

4.1.5 Wartości ze znakiem i bez znaku

Warto wspomnieć kilka właśności dwóch omawianych reprezentacji liczb całkowitych.

- Równoważność: takie samo kodowanie dla liczb nieujemnych w obydwu reprezentacjach.
- Jednoznaczność:
  - każdy układ bitów koduje jedną liczbę całkowitą,
  - każda liczba całkowita (z dopuszczalnego zakresu) ma jedno kodowanie.
- Owe kodowania można zatem odwrócić:
  - $\mathrm{U2B}(X) = \mathrm{B2U}^{-1}(X)$ — układ bitów dla danej liczby całkowitej bez znaku,
  - $\mathrm{T2B}(X) = \mathrm{B2T}^{-1}(X)$ — układ bitów dla danej liczby całkowitej ud2.

Tabela przedstawia wszystkie możliwie ciągi czterobitowe i reprezentowanie przez nie wartości liczb przy założeniu kodowania bez znaku i ud2. Kolorem zaznaczone są te wartości, które dla obydwu kodowań są wspólne.

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

4.1.6 Konwersje liczb

W języku C wiele konwersji jest wykonywanych automatycznie, bez ostrzeżenia. Są to generalnie sytuacje, w których typy zmiennych o „mniejszym” typie są rozszerzane do zmiennych o typie „większym”. Hierarchia typów, od najmniejszego do największego, jest — w pewnym uproszczeniu — taka:

```
char → unsigned char → short int → unsigned short int → int
→ unsigned int → long int → unsigned long int → long long int
  → unsigned long long int → float → double → long double
```

Niestety, typ „większy” nie zawsze jest faktycznym nadzbiorem typu „mniejszego”, co może prowadzić do utraty lub zafałszowania informacji. Przykładowo, konwersja z liczby typu `long int` na liczbę typu `float` może powodować utratę cyfr znaczących, gdyż nie wszystkie bity zmieszczą się w mantysie liczby `float`. Przykładowo, podstawienie liczby 123456789 pod zmienną typu `float` sprawi, że faktycznie zapamiętana wartość będzie równa 1234567936. Konwersje między typami całkowitymi ze znakiem i bez znaku o takim samym rozmiarze prowadzą do przekłamań opisanych poniżej.

Przykład 23.

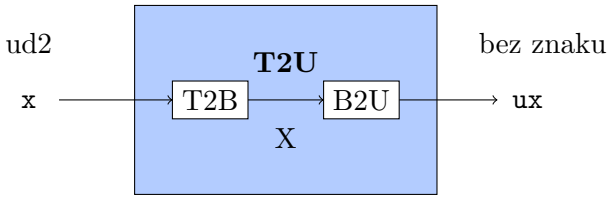
- Konwersja „ze znakiem → bez znaku”:

```
short int           x = 12345;
unsigned short int  ux = (unsigned short) x;   // konwersja
short int           y = -12345;
unsigned short int  uy = (unsigned short) y;   // konwersja
```

- Co otrzymujemy?
  - Otrzymujemy ten sam układ bitów: powyższe konwersje (zwane zwykle *rzutowaniami*) niczego nie przeliczają. Różnica jest tylko w późniejszej interpretacji liczby, a ściślej — w interpretacji najbardziej znaczącego bitu.
  - Wartości *nieujemne* pozostają bez zmian (`ux` = 12345), bowiem najbardziej znaczący bit jest tu zerem.
  - Wartości *ujemne* zmieniają się w (duże) wartości dodatnie (`uy` = 53191 przy założeniu, że słowo jest 16-bitowe), gdyż najbardziej znaczący bit jest tu równy 1, a interpretacja jego wagi zmienia się z -2^{w-1} na 2^{w-1}.

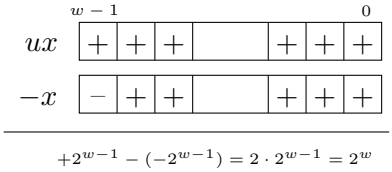
- Konwersja w przeciwną stronę, „bez znaku → ze znakiem”:
  - Sytuacja jest analogiczna: układ bitów nie zmienia się, zmienia się jedynie interpretacja bitu najbardziej znaczącego.
  - Wartości z dolnej połowy zakresu (takie, w których najbardziej znaczący bit jest zerem) mają zatem identyczną wartość jako liczba ze znakiem.
  - Wartości z górnej połowy zakresu (takie, w których najbardziej znaczący bit jest równy 1) zmieniają się w liczby ujemne. ◁

Przyjrzyjmy się bliżej konwersji liczby ze znakiem na liczbę bez znaku:



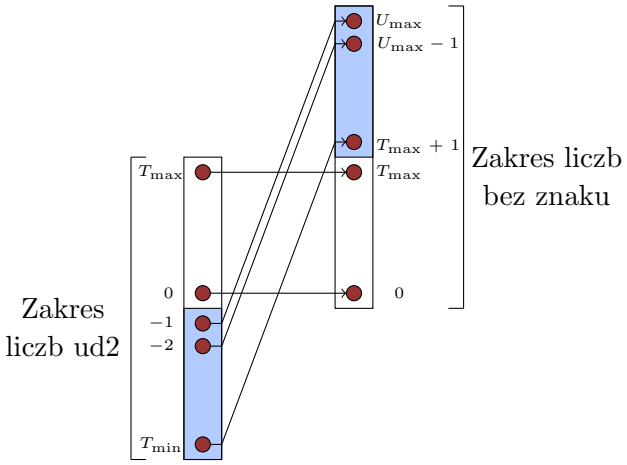
Rysunek 4.1: Zostaje ten sam układ bitów, czyli nie robimy nic

Porównajmy teraz oryginalną wartość x z otrzymaną liczbą ux, odejmując „pisemnie” $ux - x$:



$$\texttt{ux} = \begin{cases} \texttt{x}, & \texttt{x} \geqslant 0 \\ \texttt{x} + 2^w, & \texttt{x} < 0 \end{cases}$$

Jak zatem zmienia się interpretacja wartości liczby przy tej konwersji?



Przykład 24. Ponownie przedstawiamy przeliczenie ciągu bitów na wartość liczbową, tym razem dla tego samego ciągu bitów i dwóch reprezentacji. Szarą czcionką zaznaczono kolejne bity, czarną — odpowiadające im wartości z uwzględnieniem wag poszczególnych bitów w reprezentacji bez znaku i w ud2. Pokreślmy, że różnica między reprezentacjami jest wyłącznie w znaku wagi najbardziej znaczącego bitu (stąd „−/+” przy wadze 32768).

Waga		-12345		53191
1	1	1	1	1
2	1	2	1	2
4	1	4	1	4
8	0	0	0	0
16	0	0	0	0
32	0	0	0	0
64	1	64	1	64
128	1	128	1	128
256	1	256	1	256
512	1	512	1	512
1024	1	1024	1	1024
2048	1	2048	1	2048
4096	0	0	0	0
8192	0	0	0	0
16384	1	16384	1	16384
−/+ 32768	1	-32768	1	32768
Suma		-12345		53191

◁

Kilka dodatkowych uwag, które pozwalają zrozumieć zachowanie języka C, gdy używamy liczb ze znakiem i bez znaku:

- Stałe
  - Stałe domyślnie są traktowane jako liczby całkowite ze znakiem, np. zapis `0` lub `123` jest traktowany jako liczba całkowita ze znakiem.
  - Możemy wymusić interpretację stałej jako liczby bez znaku, dopisując literę U, np. `0U`, `4294967259U`.
  - Ta konwencja ma znaczenie, gdy np. porównujemy zmienną typu unsigned ze stałą −1. Niespodziewanie dla nas, stała zostanie potraktowana jako unsigned, przez co stanie się liczbą dodatnią.
- Konwersje (rzutowania)
  - Rzutowanie — jawne lub nie — działa jak U2T i T2U.

  - W poniższym kodzie podstawienie pod zmienną `tx` odpowiada funkcji U2T, podstawienie pod `uy` — funkcji T2U:

    ```
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```

  - Niejawne rzutowanie występuje przy różnych operacjach (podstawieniach, porównaniach, wywołaniach funkcji). Podstawienia w poniższym kodzie działają identycznie jak poprzednio, mimo braku jawnych rzutowań:

    ```
    tx = ux;
    uy = ty;
    ```

- Niespodzianki przy (niejawnym) rzutowaniu

  - Jeżeli wyrażenie zawiera liczby ze znakiem i bez znaku, następuje niejawne rzutowanie wartości ze znakiem do wartości bez znaku.
  - Powyższa zasada dotyczy także porównań `<`, `>`, `==`, `<=`, `>=`.

Przykład 25. Tabela pokazuje, jaka relacja (w rozumieniu języka C) zachodzi między parami wskazanych stałych i w jakiej reprezentacji będą interpretowane (po ewentualnym rzutowaniu jednej z nich, gdy ich typy nie były takie same). Przykład zakłada, że długość słowa wynosi $w = 32$.

Stała_1		Stała_2	Relacja	Interpretacja
0		0U	==	bez znaku
-1		0	<	ud2
-1		0U	>	bez znaku
2147483647		-2147483648	>	ud2
2147483647U		-2147483648	<	bez znaku
-1		-2	>	ud2
(unsigned) -1		-2	>	bez znaku
2147483647		2147483648U	<	bez znaku
2147483647	(int)	2147483648U	>	ud2

◁

Skąd biorą się niespodzianki? Przy konwersji z ud2 na reprezentację bez znaku następuje odwrócenie porządku, gdy liczby ujemne stają się „dużymi” liczbami dodatnimi.

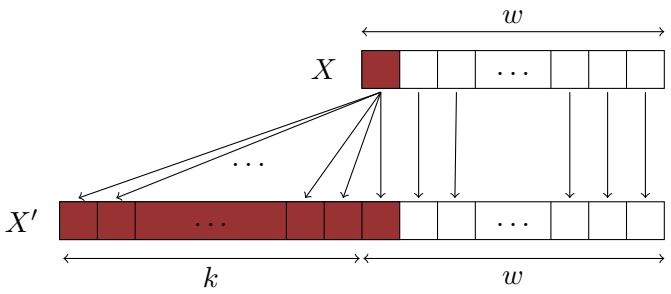
4.1.7 Powielanie bitu znaku

Rozważmy sytuację, gdy reprezentowana za pomocą ud2 liczba, zapisana w zmiennej mniejszego typu całkowitego (np. short int), jest konwertowana do typu większego (np. long int). Konieczna jest tu pewna nieoczywista operacja, polegająca na powieleniu bitu znaku (czyli najbardziej znaczącego bitu).

Załóżmy, że dana jest w-bitowa liczba, reprezentowana za pomocą ud2. Chcemy zmienić ją na liczbę „o k bitów dłuższą”, a zatem $(w + k)$-bitową. Oczywiście wartość liczby ma pozostać nie zmieniona. Sposobem na to jest zrobienie k kopii (*powielenie*) bitu znaku; nowa reprezentacja będzie zatem miała postać

$$\underbrace{x_{w-1}, \ldots, x_{w-1}}_{k}, x_{w-1}, x_{w-2}, \ldots, x_0.$$

Ilustruje to poniższy rysunek.



Przykład 26. Załóżmy poniższe deklaracje:

```
short int x  =  12345;
int ix       =  (int) x;
short int y  =  -12345;
int iy       =  (int) y;
```

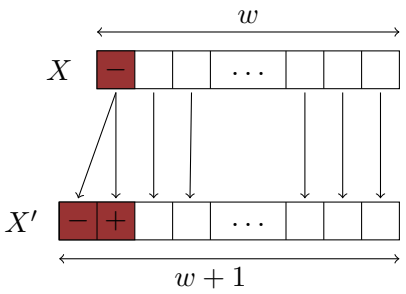
Reprezentacja ud2 poszczególnych zmiennych będzie wówczas następująca:

	dziesiętnie	ud2
x	12345	00110000 00111001
ix	12345	00000000 00000000 00110000 00111001
y	-12345	11001111 11000111
iy	-12345	11111111 11111111 11001111 11000111

Widać tu, że język C automatycznie powiela bit znaku. ◁

Twierdzimy zatem, że dla dowolnej liczby reprezentowanej w ud2 i dla dowolnego $k > 0$ wartość liczby z powielonym k-krotnie bitem znaku jest taka sama jak jej pierwotna wartość. Słuszność tej tezy uzasadnia prosta indukcja.

- Przypadek $k = 1$: jednokrotne powielenie bitu znaku zachowuje wartość liczby. Poniższy rysunek ilustruje nową interpretację bitów w powiększonej reprezentacji (X to pierwotna reprezentacja o długości w, zaś X' — reprezentacja wydłużona o jeden bit).



  Kluczowa jest tu obserwacja, że $-2^{w-1} = -2^w + 2^{w-1}$. W konsekwencji, jeśli popatrzymy na łączną wagę bitów zaznaczonych kolorem, to okaże się ona taka sama:

$$\begin{aligned} X: &\quad -2^{w-1}x_{w-1} \\ X': &\quad -2^w x_{w-1} + 2^{w-1}x_{w-1} = -2^{w-1}x_{w-1} \end{aligned}$$

  Jako że pozostałe bity ani ich wagi nie ulegają zmianie, wartość „wydłużonej” liczby jest identyczna jak pierwotnej.

- Każde kolejne powielenie oczywiście zachowuje się tak samo, zatem wartość liczby pozostaje nie zmieniona dla dowolnej wartości $k > 0$.

4.1.8 Kiedy używać liczb bez znaku?

Naturalne wydaje się pytanie o to, kiedy warto używać reprezentacji bez znaku. Jej techniczna prostota mogłaby sugerować większą efektywność i skłaniać nas do użycia typów unsigned wszędzie tam, gdzie dziedziną obliczeń są liczby naturalne. Trzeba jednak pamiętać, że w znakomitej większości przypadków nie ma różnicy w sprawności obliczeń na liczbach ze znakiem i bez znaku. Przestrogą przed pochopnym deklarowaniem zmiennych unsigned niech natomiast będzie następujący przykład pętli:

```
unsigned i;
for (i = 2; i >= 0; i--)
    a[i] += a[i+1];
```

Pętla ta nigdy się nie skończy, gdyż po osiągnięciu przez zmienną `i` wartości 0, następne zmniejszenie doprowadzi do układu bitów złożonego z samych jedynek. Układ ten, interpretowany jako liczba ze znakiem w reprezentacji ud2, oznaczałby -1, ale w reprezentacji bez znaku będzie oznaczał dużą liczbę dodatnią.

Kiedy zatem na pewno warto używać liczb bez znaku? Wymieńmy dwa zastosowania: do szczególnych rodzajów arytmetyki (arytmetyka modularna, arytmetyka wielokrotnej precyzji) oraz wówczas, gdy potrzebny jest zakres „o jeden bit większy", czyli gdy potrzebujemy wykorzystać wszystkie dostępne bity do reprezentowania liczb dodatnich.

4.1.9 Zmiana znaku przez negację i inkrementację

Załóżmy, że nie znamy odejmowania i chcemy zmienić znak liczby. Przez $\sim x$ oznaczamy (jak w języku C) negację bitów liczby x, czyli zamianę każdego zera na jedynkę i na odwrót. Mówiąc *inkrementacja*, mamy na myśli po prostu zwiększenie liczby całkowitej o jeden. Pokażemy, że w reprezentacji ud2 zachodzi równość:

Uwaga!

$$\sim x + 1 = -x.$$

Zauważmy najpierw, że dla dowolnej liczby (w istocie dla dowolnego ciągu bitów) x zachodzi $\sim x + x = 1111\ldots11_2 = -1$, przy czym -1 to oczywiście interpretacja ciągu jedynek w ud2. Własność tę ilustruje poniższe „dodawanie pisemne".

	x	1	0	0	1	1	1	0	1
+	$\sim x$	0	1	1	0	0	0	1	0
	-1	1	1	1	1	1	1	1	1

Zauważmy także, że

$$\begin{aligned}\sim x + 1 &= (\sim x + 1) + (x + (-x)) \\ &= (\sim x + x) + 1 + (-x) \\ &= -1 + 1 + (-x) \\ &= -x,\end{aligned}$$

co kończy dowód uwagi.

Przykład 27. Przyjrzyjmy się układom bitów, jeśli zastosujemy powyższą metodę do liczb 12345 i 0.

	dziesiętnie	ud2
x	12345	00110000 00111001
$\sim x$	−12346	11001111 11000110
$\sim x + 1$	−12345	11001111 11000111
0	0	00000000 00000000
~ 0	−1	11111111 11111111
$\sim 0 + 1$	0	00000000 00000000

◁

Wyliczenie, które pokazaliśmy, zapewne nie przyda się w realnym pisaniu programów. Pozwala ono natomiast lepiej zrozumieć, jak przebiegają działania na bitach.

4.1.10 Operacje na liczbach całkowitych

Jak właściwie działa arytmetyka w komputerze? Opisujemy własności faktycznych operacji wykonywanych przez komputer i wskazujemy różnice w stosunku do prawdziwej, „ludzkiej" arytmetyki.

Dodawanie bez znaku

Standardowe dodawanie wykonywane przez komputer pomija przeniesienie. W istocie jest to więc dodawanie $\mod 2^w$, gdzie w jest długością słowa. Oznaczmy je przez $\texttt{UAdd}_w$. Mamy więc

$$\texttt{UAdd}_w(u, v) = (u + v) \bmod 2^w.$$

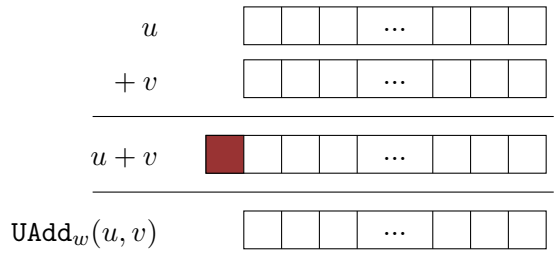
Wobec tego

$$\texttt{UAdd}_w(u, v) = \begin{cases} u + v, & u + v < 2^w, \\ u + v - 2^w, & u + v \geqslant 2^w. \end{cases}$$

Zwróćmy uwagę na liczbę bitów, biorących udział w $\texttt{UAdd}_w$:

- operandy (składniki u i v) mają w bitów,
- prawdziwa suma (z ewentualnym przeniesieniem) wymaga $w + 1$ bitów,
- wynik dodawania na komputerze pomija przeniesienie, zatem zostaje w bitów.

Sytuację tę ilustruje poniższy rysunek. Kolorem zaznaczono ewentualne przeniesienie.



Oczywiście przeniesienie nie ginie bezpowrotnie. Jest ono zapamiętywane w procesorze jako jeden z kilku *znaczników*, czyli odrębnych, jednobitowych informacji, dostępnych bezpośrednio programistom piszącym w kodzie maszynowym, ale nie w języku C. „Utrata" przeniesienia przy obliczeniach na typowych liczbach całkowitych jest wręcz charakterystyczną cechą tego i wielu innych języków.

Własności matematyczne liczb w reprezentacji bez znaku wraz z dodawaniem $\texttt{UAdd}_w$ (przy ustalonej długości słowa w) najprościej scharakteryzować mówiąc, że stanowią one grupę przemienną. Zauważmy bowiem, że: *

- zbiór ów jest zamknięty na dodawanie, $0 \leqslant \texttt{UAdd}_w(u,v) \leqslant 2^w - 1$,
- dodawanie jest przemienne, $\texttt{UAdd}_w(u,v) = \texttt{UAdd}_w(v,u)$,
- dodawanie jest łączne, $\texttt{UAdd}_w(t, \texttt{UAdd}_w(u,v)) = \texttt{UAdd}_w(\texttt{UAdd}_w(t,u), v)$,
- 0 jest elementem neutralnym dodawania, $\texttt{UAdd}_w(u,0) = u$,
- każdy element ma element przeciwny, $\texttt{UAdd}_w(u, \texttt{UInv}_w(u)) = 0$, gdzie $\texttt{UInv}_w(0) = 0$ oraz $\texttt{UInv}_w(u) = 2^w - u$ dla $u > 0$.

Dodawanie ze znakiem (w ud2)

Dodawanie ze znakiem oznaczmy przez $\texttt{TAdd}_w$ i zdefiniujmy jako działanie na w-bitowych liczbach w reprezentacji ud2, które wykonuje takie same operacje na bitach jak $\texttt{UAdd}_w$. W sensie technicznym jest to zatem identyczne działanie, zaś różnica polega jedynie na interpretacji argumentów i wyniku jako liczb reprezentowanych w ud2. Liczba bitów, biorących udział w $\texttt{TAdd}_w$, jest oczywiście taka sama jak w przypadku $\texttt{UAdd}_w$:

- operandy (składniki u i v) mają w bitów,
- prawdziwa suma wymaga $w+1$ bitów,
- wynik dodawania na komputerze pomija przeniesienie, zatem zostaje w bitów.

Uwaga: mówimy tu o przeniesieniu w sensie technicznym, czyli o dodatkowym bicie równym 1, który może pojawić się wskutek wykonania obliczeń na najbardziej znaczących bitach w-bitowych słów. Nie należy mu jednak przypisywać takiej samej interpretacji jak w dodawaniu liczb bez znaku.

Poniższy przykład pokazuje, że na poziomie procesora nie ma znaczenia, czy składniki są typu `int`, czy `unsigned int` (oczywiście przy tym samym rozmiarze).

Przykład 28. Załóżmy, że wykonaliśmy kod opisany w ramce.

```
int s, t, u, v;
u = ...;    // tu podstawiamy dowolną wartość
v = ...;    // jw.
s = (int)((unsigned)u + (unsigned)v);  // wykonujemy UAdd
t = u + v;  // wykonujemy TAdd
```

Okazuje się, że w każdym przypadku wartości zmiennych `s` i `t` będą identyczne. ◁ *

Jak zatem widać, nie ma odrębnego dodawania i odejmowania dla typów ze znakiem i bez znaku: wykonujemy dokładnie te same operacje na bitach. Różnica jest w interpretacji wyniku. Procesor w czasie takich operacji ustawia (wspomniane już wcześniej) znaczniki, które są przydatne w interpretacji wyniku, np. *znacznik przeniesienia* ma sens tylko dla liczb bez znaku. Dla liczb ze znakiem * praktycznie dowolne dwie liczby ujemne ustawią znacznik przeniesienia, który w tym przypadku nie jest interesujący. Interesujący będzie natomiast *znacznik nadmiaru* (lub *przepełnienia*), oznaczający, że wynik operacji ze znakiem nie zmieścił się w możliwym do reprezentowania zakresie. Trzeci typowy znacznik procesora, *znacznik znaku*, to po prostu powielony najbardziej znaczący bit wyniku — ma sens tylko wtedy, gdy traktujemy liczby jako liczby ze znakiem. Czwarty znacznik, *znacznik zera*, ma sens w obu interpretacjach. *

Z poziomu języka C znaczniki procesora nie są bezpośrednio dostępne. Można je sprawdzać w sposób programowy, badając wyniki obliczeń, lub napisać fragment programu w kodzie maszynowym.

Przykład 29. Rozważmy parę prostych przykładów dodawania dodatnich liczb w arytmetyce ze znakiem przy $w = 4$. Rozpisane w prawej kolumnie reprezentacje z wagami bitów służą przypomnieniu tego, że w ud2 najbardziej znaczący bit ma wagę ujemną. W rezultacie dodawanie $7+1 = 8$

prowadzi do *nadmiaru*, czyli wyniku reprezentującego inną liczbę niż wynik zwykłego dodawania. Zauważmy, że nadmiar powstaje tylko wówczas, gdy „prawdziwy" wynik nie daje się reprezentować w słowie o założonej długości w. Możemy wyróżnić *nadmiar dodatni*, gdy wynik ten jest większy od największej reprezentowalnej liczby (jak w tym przykładzie), i *nadmiar ujemny* — gdy jest mniejszy od najmniejszej reprezentowalnej. Zauważmy ponadto, że dodawanie dwóch liczb nieujemnych nigdy nie prowadzi do (technicznego) przeniesienia poza w bitów. *

- $1 + 1 = 2$

$$\begin{array}{rcccc} & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 \\ \hline & 0 & 0 & 1 & 0 \end{array} \qquad \begin{array}{cccccc} (0 & 0 & 0 & 1)_2 & = & 1 \\ -2^3 & 2^2 & 2^1 & 2^0 & & \\ \hline (0 & 0 & 1 & 0)_2 & = & 2 \\ -2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

- $7 + 1 = 8$

$$\begin{array}{rcccc} & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 0 & 1 \\ \hline & 1 & 0 & 0 & 0 \end{array} \qquad \begin{array}{cccccc} (0 & 1 & 1 & 1)_2 & = & 7 \\ -2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

- $1 + 6 = 7$

$$\begin{array}{rcccc} & 0 & 0 & 0 & 1 \\ + & 0 & 1 & 1 & 0 \\ \hline & 0 & 1 & 1 & 1 \end{array} \qquad \begin{array}{cccccc} (0 & 1 & 1 & 0)_2 & = & 6 \\ -2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

◁

Przykład 30. Rozważmy teraz dodawanie liczb ujemnych, ponownie przy $w = 4$. Szare jedynki po lewej stronie wyników obrazują techniczne przeniesienie, które pomijamy i które nie ma wpływu na poprawność wyniku. Zwróćmy jednak uwagę na dodawanie $-8 + (-8)$, którego wynik nie mieści się w czterobitowej reprezentacji ud2, stanowiąc przykład nadmiaru ujemnego.

- $-1 + (-1) = -2$

$$\begin{array}{rcccc} & 1 & 1 & 1 & 1 \\ + & 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 & 0 \end{array} \qquad \begin{array}{cccccc} (1 & 1 & 1 & 1)_2 & = & -1 \\ -2^3 & 2^2 & 2^1 & 2^0 & & \\ \hline (1 & 1 & 1 & 0)_2 & = & -2 \\ -2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

- $-8 + (-8) = -16$

$$\begin{array}{rcccc} & 1 & 0 & 0 & 0 \\ + & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array} \qquad \begin{array}{cccccc} (1 & 0 & 0 & 0)_2 & = & -8 \\ -2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

- $-2 + (-6) = -8$

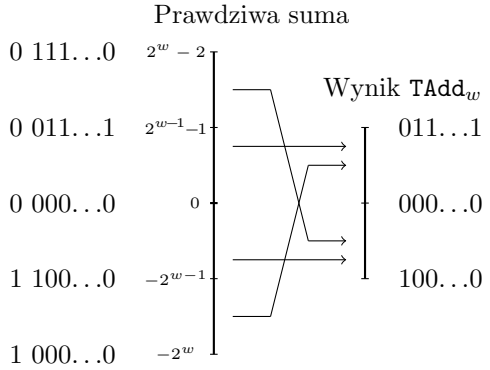
$$\begin{array}{rcccc} & 1 & 1 & 1 & 0 \\ + & 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 & 0 \end{array} \qquad \begin{array}{cccccc} (1 & 0 & 1 & 0)_2 & = & -6 \\ -2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

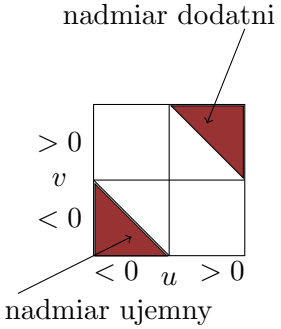
◁

Pora na podsumowanie własności dodawania ze znakiem. Przede wszystkim określmy, jaki dokładnie jest wynik dodawania w-bitowych liczb całkowitych ze znakiem. Przypomnijmy, że $T_{\min}$ i $T_{\max}$ oznaczają najmniejszą i największą reprezentowalną w ud2 liczbę, czyli $T_{\min} = -2^{w-1}$ oraz $T_{\max} = 2^{w-1} - 1$.

$$\texttt{TAdd}_w(u,v) = \begin{cases} u+v+2^{w-1}, & u+v < T_{\min} \quad \text{(nadmiar ujemny)} \\ u+v, & T_{\min} \leqslant u+v \leqslant T_{\max} \\ u+v-2^{w-1}, & T_{\max} < u+v \quad \text{(nadmiar dodatni)} \end{cases}$$

Gdy pojawia się nadmiar, wynik $\texttt{TAdd}_w$ „przeskakuje" o 2^{w-1} w górę lub w dół, w zależności od tego, czy wystąpił nadmiar ujemny, czy dodatni. Zjawisko to jest oczywiście odpowiednikiem redukcji $\bmod 2^w$ przy dodawaniu bez znaku. Odwzorowanie zakresów prawdziwych wyników w zakresy wyników $\texttt{TAdd}_w$ ilustruje lewy diagram poniżej. Zauważmy, że prawdziwy wynik może sięgać od $2T_{\min} = -2^w$ do $2T_{\max} = 2^w - 2$. Odpowiadające takim wynikom układy bitów są zapisane po lewej stronie wraz z bitem przeniesienia. Prawy rysunek pokazuje natomiast, dla jakich wartości argumentów u i v pojawia się nadmiar.





Jak wykryć nadmiar przy dodawaniu ze znakiem? Zauważmy, że nadmiar występuje wówczas, gdy argumenty mają ten sam znak i jest on różny od znaku wyniku. Innymi słowy, zachodzi następująca własność:

Uwaga! Niech s będzie wynikiem dodawania $\texttt{TAdd}_w(u,v)$. Wówczas

- jeśli $u, v < 0$ oraz $s \geqslant 0$, to mamy nadmiar ujemny,
- jeśli $u, v \geqslant 0$ oraz $s < 0$, to mamy nadmiar dodatni.

W konsekwencji, wystąpienie nadmiaru można stwierdzić wyliczając w języku C wyrażenie

```
(u < 0 == v < 0) && (u < 0 != s < 0).
```

Mnożenie

Mnożenie jest nieco bardziej skomplikowane od dodawania. Oprócz złożoności samego działania pojawia się kwestia ilości pamięci, potrzebnej do zapisania wyniku, bowiem iloczyn dwóch liczb w-bitowych może wymagać $2w$ bitów. Procesory, w zależności od wariantu mnożenia, mogą obliczać pełny wynik lub tylko jego „dolną połowę", czyli tę część, która mieści się w słowie o długości w bitów. Pozwala to zaoszczędzić nieco czasu lub pamięci w sytuacjach, gdy pełnego wyniku nie potrzebujemy. Drugą problematyczną kwestią jest rozróżnienie mnożenia liczb ze znakiem i bez znaku. Inaczej niż przy dodawaniu, tu na ogół występują dwie odrębne operacje procesora.

Dla lepszego zrozumienia, dlaczego zwykły algorytm mnożenia nie jest odpowiedni do pełnego mnożenia liczb reprezentowanych w ud2, przyjrzyjmy się przykładom dla $w = 4$. W każdym z przykładów pierwszy czynnik jest ujemny, zaś drugi — dodatni. Szare cyfry to nadmiarowe bity,

które chcielibyśmy odrzucić (umownie określmy je „przeniesieniem"). Cyfry zaznaczone na czerwono pokazują, że domniemany bit znaku „nie działa" tu właściwie. Zauważmy jednak, że nawet nieprawidłowe wyniki nie są całkiem złe: ich dolna połowa przystaje do prawidłowego wyniku mod 2^w.

Przykład 31.

- $(-1) \cdot 1 = -1$

				1	1	1	1	
			·	0	0	0	1	
				1	1	1	1	
			0	0	0	0		
		0	0	0	0			
+	0	0	0	0				
	0	0	0	1	1	1	1	

- $(-3) \cdot 3 = -9$

				1	1	0	1	
			·	0	0	1	1	
				1	1	0	1	
			1	1	0	1		
		0	0	0	0			
+	0	0	0	0				
	0	1	0	0	1	1	1	

- $(-2) \cdot 4 = -8$

				1	1	1	0	
			·	0	1	0	0	
				0	0	0	0	
			0	0	0	0		
		1	1	1	0			
+	0	0	0	0				
	0	1	1	1	0	0	0	

- $(-8) \cdot 8 = -64$

				1	0	0	0	
			·	1	0	0	0	
				0	0	0	0	
			0	0	0	0		
		0	0	0	0			
+	1	0	0	0				
	1	0	0	0	0	0	0	

◁

Policzmy, jakie są możliwe zakresy wartości prawdziwych iloczynów dla argumentów o długości w bitów: *

- dla liczb bez znaku iloczyn jest nieujemny, a największą możliwą wartością jest

$$U_{\max}^2 = (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1,$$

  zatem wynik może mieć do $2w$ bitów,

- najmniejszą możliwą wartością iloczynu dla liczb ud2 jest

$$T_{\min} T_{\max} = (-2^{w-1})(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1},$$

  zatem wynik może mieć do $2w - 1$ bitów,

- największą możliwą wartością iloczynu dla liczb ud2 jest

$$T_{\min}^2 = (-2^{w-1})^2 = 2^{2w-2},$$

  zatem wynik może mieć do $2w - 1$ bitów.

Zachowanie pełnych wyników wymaga dwukrotnego powiększania słowa przy każdym mnożeniu. Czyni się tak w obliczeniach „wielokrotnej precyzji", gdy pełne wyniki są faktycznie niezbędne. Tak realizowane mnożenie nazywa się niekiedy *rozszerzającym*. Najczęściej jednak wykonujemy mnożenie przy założeniu, że wynik zmieści się w słowie o tej samej długości co argumenty. Takie założenie przyjmowane jest w większości języków programowania (w tym C); przyjmujemy je również w dalszej części rozdziału. Taka realizacja mnożenia nosi nazwę *nierozszerzającej*.

Mnożenie bez znaku

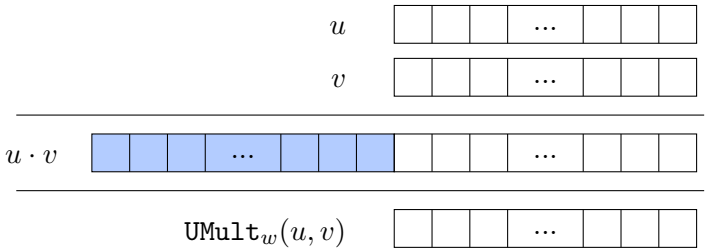
Oznaczmy przez $\texttt{UMult}_w$ nierozszerzające mnożenie liczb bez znaku o długości w bitów, które oblicza wynik o długości w bitów. Jest to więc w istocie mnożenie $\bmod 2^w$:

$$\texttt{UMult}_w(u, v) = (uv) \bmod 2^w.$$

Podsumujmy liczbę bitów, biorących udział w tej operacji:

- operandy (czynniki u i v) mają w bitów,
- prawdziwy iloczyn wymaga $2w$ bitów,
- wynik mnożenia $\texttt{UMult}_w$ pomija w bitów przeniesienia, zatem zostaje w bitów.

Sytuację tę ponownie ilustruje rysunek, na którym kolorem zaznaczono przeniesienia.



Mnożenie ze znakiem

Mnożenie liczb ze znakiem, które w wariancie nierozszerzającym oznaczmy przez $\texttt{TMult}_w$, pod względem liczby biorących w nim udział bitów jest oczywiście podobne do $\texttt{UMult}_w$. Wynik możemy zaś opisać następująco:

- dla argumentów, dla których prawdziwy iloczyn mieści się w reprezentowalnym przedziale (od $T_{\min}$ do $T_{\max}$), otrzymujemy poprawny wynik $\texttt{TMult}_w(u, v) = uv$,
- w przeciwnym przypadku otrzymujemy wynik, który przystaje do poprawnego $\bmod 2^w$, a ponadto procesor ustawia znacznik nadmiaru.

Na poziomie bitów mnożenie $\texttt{TMult}_w$ działa więc identycznie jak $\texttt{UMult}_w$.

Inne działania

Nie mamy tu miejsca, by omawiać szczegółowo inne działania na liczbach całkowitych. Przede wszystkim są to odejmowanie i dzielenie. O ile odejmowanie jest algorytmicznie bardzo podobne do dodawania, to dzielenie jest istotnie trudniejsze niż mnożenie i zazwyczaj procesory wykonują je znacznie wolniej. *

Warto w tym kontekście wspomnieć o jeszcze jednej operacji: o *przesunięciu bitowym w prawo* lub *w lewo*. Przesunięcia wydają się intuicyjnie proste, ale należy zwrócić uwagę na przesunięcia w prawo, wykonywane na liczbach ze znakiem. Są one zwane *arytmetycznymi* i oprócz samego przesunięcia bitów wykonują powielenie bitu znaku. Przesunięcia w prawo, wykonywane na liczbach bez znaku, są określane jako *logiczne* i nie powielają bitu znaku, a zwolniony bit wypełniają po prostu zerem (podobnie jak przesunięcia w lewo).

Definicja 4.1.1. Niech X będzie ciągiem bitów o długości w, czyli zgodnie z przyjętą konwencją $X = x_{w-1}x_{w-2}\ldots x_1x_0$. Przesunięcia bitowe ciągu X określamy następująco:

- przesunięcie logiczne X w prawo: $0x_{w-1}\ldots x_2x_1$,
- przesunięcie arytmetyczne X w prawo: $x_{w-1}x_{w-1}\ldots x_2x_1$,
- przesunięcie X w lewo: $x_{w-2}x_{w-3}\ldots x_00$.

Gdy mówimy o przesunięciu jednego z powyższych rodzajów *o k bitów*, mamy na myśli k-krotne powtórzenie odpowiedniego przesunięcia.

Przesunięcie zmiennej `x` o `k` bitów w lewo lub w prawo można zapisać w języku C jako `x << k` oraz `x >> k`, przy czym przesunięcie w prawo jest wykonywane jako arytmetyczne lub logiczne w zależności od tego, czy zmienna `x` jest ze znakiem, czy bez znaku. Użyteczność tego działania bierze się m.in. z możliwości wykorzystania go do wykonania mnożenia lub dzielenia przez potęgę dwójki zarówno dla liczb bez znaku, jak i ze znakiem (oczywiście należy pamiętać o możliwym przekroczeniu zakresu reprezentowalnych liczb, tak samo jak przy zwykłym mnożeniu):

$$\begin{aligned} x \ \texttt{<<}\ k &= x \cdot 2^k, \\ x \ \texttt{<<}\ k &= \lfloor x/2^k \rfloor. \end{aligned}$$

Działanie takie jest zazwyczaj szybsze niż odpowiednie jawne mnożenie i znacznie szybsze niż dzielenie. Dodajmy, że kompilatory języków programowania potrafią tę własność wykorzystać i w dających się rozpoznać sytuacjach same generują kod z użyciem przesunięć zamiast mnożenia i dzielenia.

4.1.11 Własności arytmetyki komputerowej

Podsumujmy krótko własności arytmetyki komputerowej, uwzględniając tym razem obydwa podstawowe działania: dodawanie i mnożenie. Czytelnika, szukającego definicji używanych tu pojęć z dziedziny algebry, odsyłamy do podręcznika [2].

Wniosek 4.1.1. *Niech w będzie ustaloną długością słowa. Liczby bez znaku o długości w bitów z dodawaniem $\texttt{UAdd}_w$ i mnożeniem $\texttt{UMult}_w$ tworzą pierścień przemienny.* *

Zauważmy, że dla dowolnych w-bitowych liczb t, u, v zachodzi:

- zbiór ten z dodawaniem stanowi grupę przemienną,
- zbiór jest zamknięty na mnożenie, $0 \leqslant \texttt{UMult}_w(u, v) \leqslant 2^w - 1$,
- mnożenie jest przemienne, $\texttt{UMult}_w(u, v) = \texttt{UMult}_w(v, w)$,
- mnożenie jest łączne, $\texttt{UMult}_w(t, \texttt{UMult}_w(u, v)) = \texttt{UMult}_w(\texttt{UMult}_w(t, u), v)$,
- mnożenie ma element neutralny, $\texttt{UMult}_w(u, 1) = \texttt{UMult}_w(1, u) = u$,
- mnożenie jest rozdzielne względem dodawania,

$$\texttt{UMult}_w(t, \texttt{UAdd}_w(u, v)) = \texttt{UAdd}_w(\texttt{UMult}_w(t, u), \texttt{UMult}_w(t, v)).$$

Analogiczne własności można zapisać dla liczb ze znakiem, co prowadzi do podobnego wniosku:

Wniosek 4.1.2. *Niech w będzie ustaloną długością słowa. Liczby ze znakiem o długości w bitów z dodawaniem $\texttt{TAdd}_w$ i mnożeniem $\texttt{TMult}_w$ tworzą pierścień przemienny.* *

Nietrudno zauważyć, że liczby bez znaku z działaniami $\texttt{UAdd}_w$ i $\texttt{UMult}_w$, jak również liczby ze znakiem z działaniami $\texttt{TAdd}_w$ i $\texttt{TMult}_w$, tworzą pierścienie izomorficzne z typowym pierścieniem reszt modulo 2^w. Warto też wskazać istotną różnicę we własnościach porządkowych w stosunku do zwykłych liczb całkowitych. O ile dla zwykłych liczb zachodzą własności:

$$\begin{aligned} u > 0 \ &\Rightarrow\ u + v > v, \\ u > 0,\ v > 0 \ &\Rightarrow\ uv > 0, \end{aligned}$$

to w arytmetyce komputerowej nie są one ogólnie prawdziwe. Pokazują to poniższe przykłady. *

$$\begin{aligned} \texttt{TAdd}_w(T_{\max}, 1) &= T_{\min}, \\ \texttt{TMult}_{16}(12345, 30000) &= -23191. \end{aligned}$$

4.2 Reprezentacja zmiennopozycyjna

4.2.1 Zagadki C-owe

Na początek ponownie przedstawiamy kilka zagadek, odnoszących się do działań na liczbach reprezentowanych zmiennopozycyjnie w języku C. Załóżmy deklaracje takie jak w ramce. Załóżmy ponadto, że zmienne `f` i `d` zawierają poprawne, zwykłe wartości (a nie np. nieskończoność). Dla każdego wyrażenia należy albo uzasadnić, że jest prawdziwe dla dowolnych argumentów, albo wyjaśnić, dlaczego prawdziwe nie jest.

```
int x = ...;      // tu podstawiamy dowolną wartość
float f = ...;    // tu podstawiamy dowolną zwykłą wartość
double d = ...;   // jw.
```

```
   x == (int)(float)x
  x == (int)(double)x
f == (float)(double)f
       d == (float)d
          f == -(-f)
        2/3 == 2/3.0
            d < 0.0  ⇒  (d*2) < 0.0
              d > f  ⇒  -f > -d
          d*d >= 0.0
(d + f) - d == f
```

4.2.2 Reprezentacja dwójkowa ułamków

Zanim przedstawimy istotę reprezentacji zmiennopozycyjnej, wyjaśnijmy najpierw podstawowe zasady reprezentacji liczb niecałkowitych. Najprostszym sposobem jest rozszerzenie reprezentacji bez znaku o bity, których wagi będą ujemnymi potęgami dwójki. Owe dodatkowe bity, zapisane na prawo od „przecinka dwójkowego", będą więc reprezentowały część ułamkową liczby. Uściślijmy: niech $X = x_{w-1} \dots x_1 x_0$ będzie ciągiem bitów o długości w, zaś $k \in \{0, \dots, w\}$ niech będzie ustaloną *liczbą bitów po przecinku*. Wówczas ciąg ten reprezentuje liczbę

$$\mathrm{B2U}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^{i-k}.$$

Zwróćmy uwagę, że liczba bitów po przecinku jest ustalona „poza" ciągiem X, tj. sam ciąg nie zawiera informacji o k. Oczywiście zapisując liczbę na kartce, wartość k zaznaczamy stawiając w odpowiednim miejscu przecinek lub kropkę. *

*

Przykład 32. Przykłady ułamków zapisanych binarnie (z różnymi liczbami bitów po przecinku):

$$\begin{aligned}
\tfrac{1}{16} &= 0.0001_2,\\
\tfrac{5}{16} &= 0.0101_2,\\
\tfrac{63}{64} &= 0.111111_2,\\
2\tfrac{7}{8} &= 10.111_2,\\
5\tfrac{5}{8} &= 101.101_2,\\
7\tfrac{3}{4} &= 111.11_2.
\end{aligned}$$

Policzmy, np.:

$$(\underbrace{1}_{4}\underbrace{1}_{2}\underbrace{1}_{1}.\underbrace{1}_{\frac{1}{2}}\underbrace{1}_{\frac{1}{4}})_2 = 4+2+1+\frac{1}{2}+\frac{1}{4} = 7\frac{3}{4}.$$

◁

Przytoczmy kilka własności ułamków reprezentowanych binarnie. *

- Podobnie jak w przypadku liczb całkowitych dzielenie przez 2 można wykonać za pomocą przesunięcia w prawo, zaś mnożenie przez 2 — za pomocą przesunięcia w lewo. *
- Liczby zapisane w postaci $0.111111\ldots1_2$ mają wartość „nieco poniżej" 1, jako że

  $$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^i}\to 1 \quad \text{przy } i\to\infty.$$

  Będziemy pisali $1.0-\varepsilon$, mając na myśli skończoną sumę powyższych składników („tyle jedynek, ile się zmieściło").
- Nie wszystkie liczby da się przedstawić w opisany sposób. Dokładnie można przedstawić tylko liczby postaci $x/2^k$, gdzie x i k są całkowite. Liczby wymierne, które nie są powyższej postaci, można przedstawić z użyciem powtarzającego się okresowo układu bitów. W poniższych przykładach układ ten jest zapisany w nawiasach kwadratowych: *

  $$\begin{aligned}
  \tfrac{1}{3} &= 0.[01]_2,\\
  \tfrac{1}{5} &= 0.[0011]_2,\\
  \tfrac{1}{10} &= 0.0[0011]_2.
  \end{aligned}$$

4.2.3 Definicja reprezentacji zmiennopozycyjnej

Podstawową ideą reprezentacji zmiennopozycyjnej jest przechowywanie nie tylko kolejnych bitów liczby, ale i położenia przecinka (czyli liczby k, która w definicji funkcji B2U oznaczała w istocie dodatkowy czynnik 2^{-k}). W dotychczasowej reprezentacji położenie przecinka było ustalone (poprzez * umowę zewnętrzną wobec samej reprezentacji), stąd reprezentacja taka bywa nazywana *stałoprzecinkową* lub *stałopozycyjną*. W reprezentacji *zmiennopozycyjnej* położenie przecinka będzie jawnie zapisane w reprezentacji liczby. Dodatkowo znak liczby będziemy przechowywali jako oddzielny bit.

Każdą liczbę rzeczywistą różną od zera można przedstawić jako iloczyn

$$(-1)^s \cdot M \cdot 2^E,$$

gdzie

- $s \in \{0,1\}$ jest bitem znaku,
- *mantysa* M jest liczbą z przedziału $[1,2)$, *

- *cecha* E jest liczbą całkowitą.

Zauważmy, że takie przedstawienie określa wzajemnie jednoznaczne odwzorowanie między niezerowymi liczbami rzeczywistymi a trójkami (s, M, E), przy czym mantysę można interpretować jako * cyfry znaczące, zaś cechę — jako rząd wielkości liczby (lub położenie przecinka).

Reprezentacja zmiennopozycyjna (którą będziemy skrótowo oznaczali przez *zp*) polega na przedstawieniu liczby za pomocą trójki (s, M, E), a następnie zakodowaniu tej trójki w ustalony sposób. Ów konkretny sposób kodowania musi ustalać m.in. liczbę bitów wykorzystywanych do zapisania mantysy i cechy oraz sposób reprezentacji cechy jako liczby całkowitej ze znakiem. Trzeba także ustalić sposób kodowania zera i ewentualnych dodatkowych wartości spoza zbioru liczb rzeczywistych (np. nieskończoności).

Wszystkie powyższe szczegóły określa standard IEEE 754, ustalony w 1985 roku przez Institute of Electrical and Electronics Engineers jako jednolity standard arytmetyki zp (ang. floating-point arithmetic). Został on powszechnie przyjęty i zastąpił istniejące wcześniej różne formaty. Jego powstanie motywowane było względami obliczeniowymi: przewiduje on np. rozbudowaną obsługę zaokrągleń, nadmiaru i niedomiaru, ale jest trudny do szybkiego zaimplementowania. Jednak dzięki * swojej uniwersalności przetrwał do dzisiaj bez większych zmian i nie widać powodów, by miał zostać zastąpiony czymś innym. Omawiając reprezentację zp w dalszej części tekstu, będziemy się odnosili * do tego właśnie standardu.

Pozostaje nam teraz ustalić szczegóły kodowania trójki (s, M, E). Standard określa je w następujący sposób.

Rozmieszczenie bitów:

- pole `s` koduje bit znaku s,
- pole `exp` koduje cechę E,
- pole `frac` koduje mantysę M.

Liczba bitów, przeznaczonych na poszczególne pola, zależy od konkretnego wariantu reprezentacji. Standard definiuje kilka wariantów, spośród których w najpowszechniejszym użyciu są następujące trzy.

Liczba bitów pól, odpowiednio, `exp` i `frac` w typowych wariantach reprezentacji zp:

- *pojedyncza precyzja* (`single, float`): 8 i 23 — razem 32 bity,
- *podwójna precyzja* (`double`): 11 i 52 — razem 64 bity,
- *rozszerzona precyzja* (`extended`): 15 i 63 — tylko w procesorach Intela i pochodnych; przechowywane w 80 bitach (1 bit jest niewykorzystany).

Pole `s` jest zawsze jednobitowe.

Zanim określimy konkretny sposób wykorzystania bitów w polach `s`, `exp` i `frac`, wymieńmy różne rodzaje wartości, jakie możemy reprezentować. Oprócz zwykłych wartości liczbowych chcemy bowiem mieć możność przechowywania pewnych wartości specjalnych.

Co właściwie reprezentujemy?

- *wartości znormalizowane*, czyli liczby odpowiadające trójkom (s, E, M), zgodnie z wcześniejszą definicją,
- *małe liczby nieznormalizowane*, które odpowiadają trójkom (s, M, E) z mantysą $M \in [0, 1)$; wśród nich znajduje się zero,
- nieskończoności i *nieliczby* (*Not-a-Number*, *NaN*), służące do reprezentowania wyników błędnych lub spoza zakresu.

Niezależnie od wariantu reprezentujemy oczywiście pewien skończony podzbiór liczb rzeczywistych (a ściślej — wymiernych) oraz skończoną liczbę wartości specjalnych.

Pora już teraz na określenie sposobu wykorzystania bitów w polach `s`, `exp` i `frac` dla wartości różnego rodzaju. We wszystkich przypadkach znak kodowany jest za pomocą jednego bitu, który jest niezależny od pozostałych; po prostu 0 w polu `s` oznacza znak +, zaś 1 oznacza znak −. W szczególności, liczba zero ma dwie reprezentacje: jedną ze znakiem + i drugą ze znakiem −.

Wartości znormalizowane

Wartości znormalizowane stanowią podstawowy rodzaj danych, przechowywanych w reprezentacji zp. Określenie „znormalizowane" odnosi się do tego, że mantysa mieści się w przedziale $[1, 2)$. Bez tego założenia każda liczba miałaby wiele możliwych reprezentacji. Zawężenie mantysy do przedziału $[1, 2)$ nie tylko zapewnia jednoznaczność, ale ponadto — jak zobaczymy poniżej — pozwala zaoszczędzić jeden bit. Szczegóły reprezentacji wartości znormalizowanych są następujące:

- Pole `exp` nie składa się z samych zer ani z samych jedynek, $\texttt{exp} \neq 000\ldots0$ oraz $\texttt{exp} \neq 111\ldots1$ (same zera i same jedynki pełnią specjalne funkcje, związane z wartościami nieznormalizowanymi).

- Pole `exp` koduje cechę E z *przesunięciem* w ten sposób, że

$$E = \mathrm{B2U}(\texttt{exp}) - p,$$

gdzie $p = 2^{|\texttt{exp}|-1} - 1$, a $|\texttt{exp}|$ oznacza liczbę bitów pola `exp`. Innymi słowy, w polu `exp` zapisujemy wartość cechy przesuniętą o połowę dostępnego zakresu. Jest to pewien sposób reprezentacji liczby ze znakiem, inny niż ud2, stosowany tu ze względu na łatwość porównywania liczb.

Dla liczb pojedynczej precyzji przesunięcie wynosi zatem $p = 127$, a zakres możliwych cech E to $-126, \ldots, 127$ (gdyż zakres wartości w polu `exp` to $1, \ldots, 254$). Dla liczb podwójnej precyzji $p = 1023$, a zakres E to $-1022, \ldots, 1023$.

- Mantysa kodowana jest za pomocą pola `frac` w ten sposób, że interpretujemy ją jako $M = \mathrm{B2U}(1.xxx\ldots x)$, gdzie $xxx\ldots x$ oznacza bity pola `frac`.

Mówimy, że mantysa kodowana jest *z domyślną jedynką*, gdyż owej jedynki przed przecinkiem nie musimy przechowywać. Ta konwencja pozwala zaoszczędzić jeden bit reprezentacji, a przy tym gwarantuje znormalizowany zapis mantysy z przedziału $[1, 2)$.

Zauważmy, że mantysa jest najmniejsza przy $\texttt{frac} = 000\ldots0$ (wówczas $M = 1$) i największa przy $\texttt{frac} = 111\ldots1$ ($M = 2 - \varepsilon$).

Przykład 33. Przykład reprezentacji zp pojedynczej precyzji dla liczby 12345 ($|\texttt{exp}| = 8$, $|\texttt{frac}| = 23$).

- Zacznijmy od zwykłej reprezentacji bez znaku:

$$12345_{10} = 11000000111001_2 = 1.1000000111001_2 \cdot 2^{13}.$$

Druga równość to normalizacja, która polega w istocie na przesunięciu przecinka za pierwszą jedynkę liczby.

- Zapiszmy mantysę i pole `frac`:

$$M = 1.\underline{1000000111001}_2$$
$$\texttt{frac} = \underline{1000000111001}0000000000$$

  Podkreślone bity to cyfry mantysy po przecinku. Na użytek kodowania w polu `frac` trzeba je dopełnić zerami do pełnej długości tego pola.

- Zapiszmy cechę i pole `exp`, wiedząc że przesunięcie $p = 127$:

$$E = 13$$
$$\texttt{exp} = \text{U2B}(13 + 127) = \text{U2B}(140) = 10001100.$$

Reprezentacja zp liczby 12345 wygląda zatem następująco:

reprezentacja zp	0100	0110	0100	0000	1110	0100	0000	0000
znak +	0							
`exp` = 140	100	0110	0					
12345			**1**100	0000	1110	01		

Cyfra 1 zapisana pogrubioną czcionką to „domyślna jedynka”: pierwsza cyfra mantysy, która nie pojawia się w faktycznej reprezentacji.

◁

Wartości nieznormalizowane

Wartości nieznormalizowane obejmują małe liczby, nieskończoności i nieliczby. Omówimy je po kolei.

Małe liczby nieznormalizowane

Przyjrzyjmy się następującemu wyliczeniu. Najmniejszą na moduł wartość znormalizowaną otrzymujemy, gdy pole `frac` zawiera same zera, zaś pole `exp` — ciąg zer zakończony jedynką. Odpowiada to $M = 1$ i $E = 1 - p$, czyli liczbie 2^{1-p}. Mniejszej liczby nie możemy reprezentować w sposób znormalizowany. Określmy zatem konwencję *małych liczb nieznormalizowanych*:

- pole `exp` składa się z samych zer (ten układ bitów zarezerwowaliśmy właśnie na tę okazję),
- cechę E interpretujemy jako $1 - p$,
- mantysę interpretujemy jako $M = \text{B2U}(0.xxx\ldots x)$, gdzie $xxx\ldots x$ oznacza bity pola `frac` (domyślną jedynkę zastępujemy zerem).

Zauważmy, że $\texttt{exp} = 000\ldots0$ oznacza tę samą cechę co $\texttt{exp} = 000\ldots1$, ale mantysa ma teraz przed przecinkiem zero. Dzięki temu możemy zapisać jeszcze mniejsze liczby poprzez umieszczenie wiodących zer w mantysie. Innymi słowy, porzucamy normalizację, którą gwarantowała domyślna jedynka z przody mantysy, i reprezentujemy mniejsze liczby pozwalając na wiodące zera. Przykładowo:

$$0.11111111 \cdot 2^{1-p}$$
$$0.01111111 \cdot 2^{1-p}$$
$$0.00111111 \cdot 2^{1-p}$$
$$\ldots$$
$$0.00000001 \cdot 2^{1-p}$$

Mniejsze liczby zapisujemy tu za cenę gorszej dokładności, gdyż wiodące zera zabierają miejsce, które mogłoby służyć do zapisania znaczących bitów. Ta własność bywa nazywana „stopniowym niedomiarem".

Zwróćmy uwagę na bardzo ważny szczególny przypadek małej liczby. Otóż układ pól `exp` i `frac` złożonych z samych zer reprezentuje liczbę zero (gdyż jest to $0 \cdot 2^{1-p}$). W zależności od pola `s` mamy przy tym dwie osobne wartości $+0$ i -0. Arytmetyka zp traktuje je tak, jak można by oczekiwać, np. typowe porównanie traktuje je jako równe, ale jednocześnie $(-1) \cdot (-0) = +0$.

Nieskończoności i nieliczby

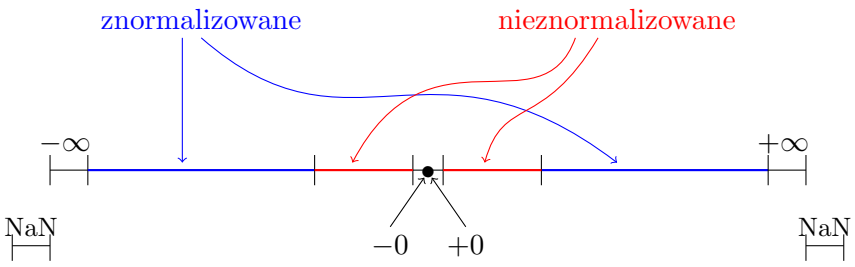
Nieskończoności (dodatnia i ujemna, czyli $+\infty$ i $-\infty$) reprezentowane są za pomocą pola `exp` złożonego z samych jedynek i pola `frac` złożonego z samych zer. Zasadniczo służą do oznaczenia nadmiaru w obliczeniach, ale pozwalają też wyrazić pewne wyniki w nieco rozszerzonej arytmetyce liczb rzeczywistych, np. $1/0 = (-1)/(-0) = +\infty$, $1/(-0) = -\infty$, $(-1)/(+\infty) = -0$. Podobnie dla dowolnej liczby x zachodzi $-\infty < x < +\infty$.

Nieliczby (Not-a-Number, NaN) reprezentują sytuacje, w których nie da się ustalić wartości, np. $\sqrt{-1}$, $\infty - \infty$. Można je więc traktować jako kody błędów w obliczeniach. Nieliczby zapisywane są za pomocą pola `exp` złożonego z samych jedynek i pola `frac` złożonego z ciągu różnego od samych zer.

Zestawienie różnych wartości w zp

W tabeli prezentujemy wszystkie możliwe rodzaje wartości, jakie można reprezentować w zp, wraz z ich charakterystycznymi układami bitów w polach `exp` i `frac`. Rysunek pod tabelą obrazuje natomiast rozmieszczenie tych wartości na osi liczb rzeczywistych. Uwaga: każda z wartości może mieć znak + lub −, zależnie od bitu w polu `s`. Układy bitów w ostatniej kolumnie tabeli ($1.xxx$, $0.xxx$, `exp`) należy rozumieć jako ich wartości w reprezentacji bez znaku.

Rodzaj wartości	`exp`	`frac`	Interpretacja
znormalizowane	$\neq 000\ldots0$, $\neq 111\ldots1$	dowolne xxx	$1.xxx \cdot 2^{\texttt{exp}-p}$
małe liczby	$000\ldots0$	dowolne xxx	$0.xxx \cdot 2^{\texttt{exp}-p}$
w tym zero	$000\ldots0$	$000\ldots0$	0
nieskończoności	$111\ldots1$	$000\ldots0$	∞
nieliczby	$111\ldots1$	$\neq 000\ldots0$	wartość błędna



Przykład 34. Przykład bardzo niewielkiej reprezentacji zp i zakresy liczb, jakie można w niej zapisać.

Rozważmy reprezentację zp o łącznej długości 8 bitów, skonstruowaną analogicznie do standardowych wariantów. Zakładamy, że pole `exp` ma 4 bity, zaś pole `frac` — 3 bity. Przesunięcie wynosi zatem $p = 2^{4-1} - 1 = 7$. Zgodnie z zasadami standardu możemy zapisywać wartości znormalizowane i nieznormalizowane, w tym nieskończoności i nieliczby. Zakresy liczb przy tych założeniach przedstawia poniższa tabela (przedstawiamy dodatnią połowę tabeli; ujemna połowa jest symetryczna). Wartości nieznormalizowane oznaczone są gwiazdką w pierwsze kolumnie.

	s	exp	frac	E	Wartość	Uwagi	
*	0	0000	000	-6	$0 \cdot 2^{-6}$	$= 0/512$	0
*	0	0000	001	-6	$\frac{1}{8} \cdot 2^{-6}$	$= 1/512$	najbliższa 0
*	0	0000	010	-6	$\frac{2}{8} \cdot 2^{-6}$	$= 2/512$	
*	...						
*	0	0000	110	-6	$\frac{6}{8} \cdot 2^{-6}$	$= 6/512$	
*	0	0000	111	-6	$\frac{7}{8} \cdot 2^{-6}$	$= 7/512$	największa nieznorm.
	0	0001	000	-6	$1 \cdot 2^{-6}$	$= 8/512$	najmniejsza znormal.
	0	0001	001	-6	$1\frac{1}{8} \cdot 2^{-6}$	$= 9/512$	
	...						
	0	0110	110	-1	$1\frac{6}{8} \cdot 2^{-1}$	$= 14/16$	
	0	0110	111	-1	$1\frac{7}{8} \cdot 2^{-1}$	$= 15/16$	najbliższa 1 od dołu
	0	0111	000	0	$1 \cdot 2^{0}$	$= 1$	
	0	0111	001	0	$1\frac{1}{8} \cdot 2^{0}$	$= 9/8$	najbliższa 1 od góry
	0	0111	010	0	$1\frac{2}{8} \cdot 2^{0}$	$= 10/8$	
	...						
	0	1110	110	7	$1\frac{6}{8} \cdot 2^{7}$	$= 224$	
	0	1110	111	7	$1\frac{7}{8} \cdot 2^{7}$	$= 240$	największa znormal.
*	0	1111	000			∞	

◁

4.2.4 Zakresy liczb i własności zp

Przytoczmy tu zakresy liczb, jakie można reprezentować w standardowych typach pojedynczej i podwójnej precyzji, czyli w typach najczęściej nazywanych `float` i `double`.

	exp	frac	Wartość liczbowa
(nieznormalizowane)			
0	00...00	00...00	0
najmniejsza `float` nieznorm.	$\underbrace{00\ldots00}_{8}$	$\underbrace{00\ldots01}_{23}$	$2^{-23} \cdot 2^{-126} \approx 1.4 \cdot 10^{-45}$
najmniejsza `double` nieznorm.	$\underbrace{00\ldots00}_{11}$	$\underbrace{00\ldots01}_{52}$	$2^{-52} \cdot 2^{-1022} \approx 4.9 \cdot 10^{-324}$
największa `float` nieznorm.	00...00	11...11	$(1-\varepsilon) \cdot 2^{-126} \approx 1.18 \cdot 10^{-38}$
największa `double` nieznorm.	00...00	11...11	$(1-\varepsilon) \cdot 2^{-1022} \approx 2.2 \cdot 10^{-308}$
(znormalizowane)			
najmniejsza `float` znorm.	$\underbrace{00\ldots01}_{8}$	$\underbrace{00\ldots00}_{23}$	$1 \cdot 2^{-126}$
najmniejsza `double` znorm.	$\underbrace{00\ldots01}_{11}$	$\underbrace{00\ldots00}_{52}$	$1 \cdot 2^{-1022}$
1	01...11	00...00	1
największa `float` znorm.	11...10	11...11	$(2-\varepsilon) \cdot 2^{127} \approx 3.4 \cdot 10^{38}$
największa `double` znorm.	11...10	11...11	$(2-\varepsilon) \cdot 2^{1023} \approx 1.8 \cdot 10^{308}$

Przybliżone zakresy i precyzja liczb zp:

- pojedyncza precyzja (`float`)

$$-3.4 \cdot 10^{38}, \ldots, 3.4 \cdot 10^{38}, \quad \text{7 cyfr dziesiętnych},$$

- podwójna precyzja (`double`)

$$-1.8 \cdot 10^{308}, \ldots, 1.8 \cdot 10^{308}, \quad \text{15/16 cyfr dziesiętnych}.$$

Na koniec rozważań o postaci reprezentacji zp, wymieńmy pewną jej cechę, która uzasadnia wybór zaskakującego sposobu zapisywania cechy w polu `exp`: liczby zp można porównywać prawie jak najzwyklejsze liczby bez znaku. Wyjaśnijmy owo „prawie". Otóż jeśli uwzględnimy najpierw bity znaku porównywanych liczb (i szczególny przypadek $+0 = -0$), to zwykłe porównanie dwóch liczb zp „bit po bicie" daje prawidłowy wynik nawet wówczas, gdy biorą w nim udział liczby nieznormalizowane.

Jeszcze jedna trywialna własność, która z punktu widzenia programowania jest bardzo użyteczna, to dogodna reprezentacja zera w postaci samych bitów zerowych. Umożliwia to np. prostą inicjalizację dużych tablic zerami (reprezentacjami liczby zero) przez wypełnienie ich samymi bitami zerowymi.

4.2.5 Operacje na liczbach zp

Zanim omówimy szczegóły podstawowych działań, przedstawmy najpierw ogólną koncepcję, która jest inna niż w przypadku liczb całkowitych. Koncepcja jest taka: policzmy najpierw dokładny wynik, a następnie dopasujmy go do rozmiaru reprezentacji. Dopasowanie oznacza zapisanie informacji o nadmiarze, jeżeli wynik ma zbyt dużą cechę, oraz zaokrąglenie, jeśli niektóre bity mantysy nie mieszczą się w polu `frac`.

Warto nadmienić, że standard zp definiuje różne sposoby zaokrąglania: w stronę zera, w dół (ku $-\infty$), w górę (ku $+\infty$) i „do parzystej". Ten ostatni sposób polega na zaokrągleniu do najbliższej wartości o żądanej dokładności, przy czym jeżeli jesteśmy dokładnie w połowie między dwiema wartościami, to zaokrąglamy tak, by najmniej znacząca cyfra była parzysta (czyli równa 0). Jest to domyślny tryb zaokrąglania, który ma najlepsze właściwości statystyczne. Inne tryby mają defekty statystyczne, np. suma dodatnich liczb może być konsekwentnie zbyt duża lub zbyt mała.

Działania omawiamy, odwołując się do pojęcia cechy i mantysy, a nie fizycznych pól `exp` i `frac`, gdyż w ten sposób łatwiej opisać istotę działań na liczbach zp.

Mnożenie zp

Zaczynamy od mnożenia, gdyż jego realizacja jest prostsza niż dodawania. Załóżmy, że mnożone liczby są postaci

$$(-1)^{s_1} \cdot M_1 \cdot 2^{E_1} \quad \text{oraz} \quad (-1)^{s_2} \cdot M_2 \cdot 2^{E_2}.$$

Wówczas wynik dokładny ma postać

$$(-1)^{s} \cdot M \cdot 2^{E},$$

gdzie:

- znak $s = s_1 \text{ xor } s_2$ (xor oznacza alternatywę rozłączną),
- mantysa $M = M_1 \cdot M_2$,
- cecha $E = E_1 + E_2$.

Tak otrzymany wynik może jednak nie spełniać zasad reprezentacji zp. Konieczne mogą być poprawki.

• Jeśli mantysa $M > 2$, należy przesunąć M o jeden bit w prawo i zwiększyć E o jeden. Zauważmy, że iloczyn mantys nigdy nie przekracza 4, zatem powyższa normalizacja nigdy nie wymaga przesunięcia dłuższego niż jeden bit.

• Jeśli cecha E jest spoza przyjętego zakresu, należy tę sytuację odpowiednio zapisać lub zasygnalizować. Zauważmy, że możemy mieć do czynienia z dwiema odmiennymi w istocie sytuacjami. Jeśli otrzymaliśmy cechę większą niż największa reprezentowalna, mówimy o nadmiarze i zapisujemy go jako $+/-\infty$. Jeśli zaś otrzymaliśmy cechę mniejszą niż najmniejsza reprezentowalna, mówimy o *niedomiarze* i (zazwyczaj) zapisujemy go po prostu jako zero. Sygnalizacja nadmiaru lub niedomiaru to odpowiednia reakcja procesora, pozwalająca programiście podjąć dodatkowe kroki.

• Jeśli mantysa M ma więcej znaczących bitów niż mieści się w polu `frac`, należy ją zaokrąglić.

Dodawanie zp

Ponownie załóżmy, że liczby są postaci

$$(-1)^{s_1} \cdot M_1 \cdot 2^{E_1} \quad \text{oraz} \quad (-1)^{s_2} \cdot M_2 \cdot 2^{E_2}.$$

Dodatkowo przyjmijmy, że $E_1 \geqslant E_2$. Wówczas dokładna suma ma postać

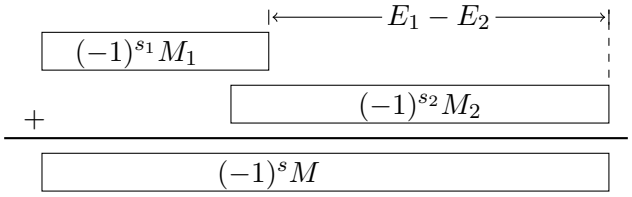
$$(-1)^s \cdot M \cdot 2^E,$$

gdzie:

- znak $s = s_1$,
- mantysa M obliczona jest zgodnie ze schematem poniżej (przesuwamy M_2 o $E_1 - E_2$ bitów w prawo i dodajemy „pisemnie" mantysy M_1 i M_2),
- cecha $E = E_1$.

Podobnie jak w przypadku mnożenia, wynik może wymagać poprawek.

- Jeśli $E_1 = E_2$, to za s należy wziąć znak liczby o większej mantysie.
- Jeśli mantysa $M = 0$, należy zapisać wynik jako zero w odpowiedniej postaci.
- Jeśli $M > 2$, należy przesunąć M o jeden bit w prawo i zwiększyć E o jeden. Ta normalizacja jest taka sama jak w przypadku mnożenia.
- Jeśli $M \in (0, 1)$, należy przesunąć M o tyle bitów w lewo, by uzyskać $M \in [1, 2)$, i o tyle samo zmniejszyć E. Taka normalizacja w ogóle nie ma okazji wystąpić przy mnożeniu.
- Jeśli cecha E jest spoza przyjętego zakresu, należy podjąć odpowiednią akcję jak przy mnożeniu.
- Jeśli mantysa M ma więcej znaczących bitów niż mieści się w polu `frac`, należy ją zaokrąglić.



Rysunek 4.2: Schemat dodawania zp — zakładamy, że $E_1 \geqslant E_2$

Zauważmy, że przesunięcie mantysy M_2 tak, by zgrać wagi jej bitów z mantysą M_1, powoduje zazwyczaj utratę bitów wysuniętych za M_1. Innymi słowy, że w wyniku musimy uwzględnić bardziej znaczące bity z M_1, a zaokrągleniu podlegają właśnie te „wystające" na długości $E_1 - E_2$. Konsekwencją jest brak łączności dodawania, co ilustruje poniższy przykład.

Przykład 35. Zadeklarujmy zmienne typu `double` jak w poniższej ramce, a następnie wypiszmy ich wartości. Najprawdopodobniej okaże się, że pierwsza z nich ma wartość 3, zaś druga — zero. Dzieje się tak dlatego, że przy dodawaniu liczby 3 do dużej na moduł liczby -10^{20} bity tej pierwszej zostają wysunięte na prawo i utracone. Jeśli jednak najpierw wykonujemy dodawanie obydwu dużych liczb, otrzymujemy zero, które zostanie potem poprawnie dodane do 3. Uwaga: kompilator może dokonać optymalizacji obliczeń, w wyniku której faktyczna kolejność dodawań ulegnie zmianie i powyższy efekt zniknie.

```
double d1 = (1e20 + (-1e20)) + 3;
double d2 = 1e20 + ((-1e20) + 3);
```

Przykład dowodzi, że w arytmetyce zp nie możemy liczyć na typowe własności arytmetyki liczb rzeczywistych. Łączności nie możemy być pewni ani w dodawaniu, ani w mnożeniu. Dodajmy, że przyczyną mogą być nie tylko zaokrąglenia, ale i nadmiar (gdy suma dwóch składników wyniesie ∞, późniejsze dodanie trzeciego składnika nie przywróci już faktycznego wyniku, nawet jeśli mieściłby się on w reprezentowanym zakresie). Inne problemy arytmetyki zp wskazujemy w poniższym podsumowaniu.

Własności arytmetyki zp

Liczby zp ustalonego rozmiaru niestety nie tworzą takich struktur algebraicznych jak liczby rzeczywiste. Oczekiwalibyśmy optymistycznie, że liczby te z dodawaniem będą tworzyły grupę przemienną, a dodatkowo z mnożeniem — pierścień przemienny. Przyjrzyjmy się, które z tych oczekiwanych własności zachodzą. Mówiąc o zbiorze liczb zp mamy na myśli zbiór liczb dających się zapisać w reprezentacji zp ustalonego rozmiaru, wraz z nieliczbami i nieskończonościami. Słowa „prawie" używamy na oznaczenie sytuacji, że dana własność zachodzi, o ile argumenty nie są nieliczbami lub nieskończonościami i o ile nie otrzymamy wyniku nieskończonego.

- Dodawanie liczb zp:
  - zbiór liczb zp jest prawie zamknięty na dodawanie,
  - dodawanie jest przemienne,
  - dodawanie nie jest w ogólności łączne,
  - dodawanie ma element neutralny (0),
  - prawie każdy element ma element przeciwny.
- Mnożenie liczb zp:
  - zbiór liczb zp jest prawie zamknięty na mnożenie,
  - mnożenie jest przemienne,
  - mnożenie nie jest w ogólności łączne,
  - mnożenie ma element neutralny (1),
  - nie każdy element ma element odwrotny,
  - mnożenie nie jest w ogólności rozdzielne względem dodawania.
- Własności porządkowe:
  - dla prawie wszystkich liczb a, b, c zachodzi $a > b \Rightarrow a + c > b + c$,

– dla prawie wszystkich liczb a, b, c zachodzi $a > b,\ c > 0 \Rightarrow a \cdot c > b \cdot c$.

Zauważmy, że pierwotną przyczyną braku niektórych własności w zbiorze liczb zp jest oczywiście skończoność reprezentacji, która ogranicza i zakres liczb, i ich dokładność. O ile w przypadku liczb całkowitych niektóre własności arytmetyki komputerowej (np. łączność) „ocaliliśmy" dzięki zawinięciu mod 2^w, to w przypadku liczb rzeczywistych takie rozwiązanie nie wchodzi w grę.

4.2.6 Liczby zp w języku C

W języku C i w wielu językach, które się z niego wywodzą, występują zwykle co najmniej dwa typy liczb zp: `float` i `double`. Są to 32- i 64-bitowe liczby zp, które określaliśmy również jako liczby pojedynczej i podwójnej precyzji.

Konwersje

Warto pamiętać, że konwersje pomiędzy typami `int` (wraz z jego odmianami), `float` i `double` mogą zmieniać wartości liczb lub prowadzić do niezdefiniowanych wyników, nawet jeśli zmienne są (pozornie) tego samego rozmiaru. Konwersje takie są w praktyce bardzo częste, rozważmy zatem możliwe przypadki.

- Konwersja `double` lub `float` do `int` obcina część ułamkową, zaokrąglając w stronę zera. Jej wynik jest zasadniczo nieokreślony, jeśli wartość jest poza zakresem liczby całkowitej — a pamiętajmy, że zakres liczb `double` lub `float` jest znacznie szerszy niż liczb całkowitych. Nie ma tutaj zawijania mod 2^w.
- Konwersja `int` do `double` jest dokładna, o ile wartość całkowita ma nie więcej znaczących bitów niż mantysa `double`, czyli 53 bity (licząc razem z domyślną jedynką). Tymczasem wiele typów całkowitych (`long int`, a często nawet zwykły `int`) ma długość 64 bitów i wartość może nie zmieścić się w mantysie bez zaokrąglenia. Mniej znaczące bity liczby całkowitej zostaną wówczas utracone wskutek zaokrąglenia.
- Konwersja `int` do `float` działa analogicznie jak konwersja `int` do `double`, ale tu ryzyko utraty znaczących bitów jest większe, gdyż mantysa `float` jest krótsza — 24 bity razem z domyślną jedynką.

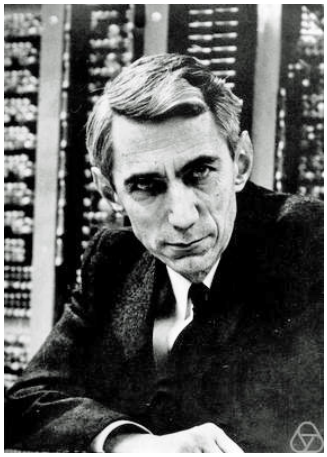
Ku przestrodze: przypadek Ariane 5

4 czerwca 1996 rakieta Ariane 5, przewożąca ładunek wart 370 milionów dolarów, eksplodowała 37 sekund po starcie[2]. Jej poprzedniczka, Ariane 4, ponad stukrotnie startowała pomyślnie. Co się stało, skoro używano tego samego oprogramowania? Otóż obliczano prędkość jako wartość typu `float` i konwertowano do 16-bitowej wartości typu `int`. Ariane 5 miała większą prędkość i owa wartość nie zmieściła się w zakresie liczby `int`.

[2] Zob. `https://en.wikipedia.org/wiki/Ariane_flight_V88`

Rozdział 5

Teoria informacji



Rysunek 5.1: Claude Elwood Shannon[1]

5.1 Miara informacji i entropia

Zajmujemy się teraz najbardziej podstawowym modelem przesyłania i mierzenia informacji, który został stworzony przez Claude'a Shannona w drugiej połowie lat 40. XX wieku. Przesyłanie informacji opisujemy za pomocą dyskretnej zmiennej losowej, którą w tym kontekście nazywamy *źródłem informacji*; jej wartości nazywamy *komunikatami*. Gdy zatem mówimy, że źródło informacji X wysyła komunikat x, oznacza to, że zmienna X przyjmuje wartość x. Będziemy też pisali $x \in X$ na oznaczenie tego, że x jest jednym z możliwych komunikatów. Zbiór możliwych komunikatów konkretnej zmiennej będziemy na ogół oznaczać przez $\{x_1, \ldots, x_n\}$, zaś ich prawdopodobieństwa — przez p_i, przy czym $p_i = P(x_i) = P(X = x_i)$ oraz $p_1 + \ldots + p_n = 1$. Kontynuując wcześniejsze nadużycia, będziemy zatem pisali $X = \{x_1, \ldots, x_n\}$. W gruncie rzeczy nie potrzebujemy teraz jednak złożonego aparatu probabilistyki; zupełnie wystarczy intuicyjne rozumienie pojęć źródła informacji i komunikatów.

Przykład 36. Rzucamy monetą, więc zbiór możliwych komunikatów to $\{\text{orzeł}, \text{reszka}\}$. Zazwyczaj zakładamy, że $P(\text{orzeł}) = P(\text{reszka}) = \frac{1}{2}$, czyli że każdy z komunikatów jest tak samo prawdopodobny. Zauważmy, że nie jest to wniosek z twierdzeń rachunku prawdopodobieństwa, a założenie, które wyraża nasze przekonanie o symetrii tej sytuacji, będące zapewne konsekwencją fizycznej symetrii monety i rozsądnej metody rzucania. ◁ *

Definicja 5.1.1. Ustalmy $r \geqslant 2$.

[1] Wikipedia, licencja `https://creativecommons.org/licenses/by-sa/2.0/de/deed.en`

- *Ilością informacji* w pojedynczym komunikacie $x_i \in X$, występującym z prawdopodobieństwem p_i, nazywamy liczbę
$$h(x_i) = \log_r \frac{1}{p_i} = -\log_r p_i.$$

- *Entropią* $H(X)$ źródła informacji X nazywamy średnią ilość informacji przypadającą na pojedynczy komunikat, czyli *

$$H(X) = \sum_{i=1}^{n} p_i h(x_i) = -\sum_{i=1}^{n} p_i \log_r p_i.$$

Uwaga: dla porządku przyjmujemy, że jeśli $p_i = 0$, to $p_i \log_r p_i = 0$.

Entropia to niepewność wystąpienia danego zdarzenia elementarnego. Jeżeli z góry wiadomo, co się stanie — nie ma niepewności.

Przykład 37. Przyjmując za podstawę logarytmu $r = 2$, tym samym ustalamy jednostkę — bit, najczęściej stosowaną jednostkę ilości informacji.

1. Rzucamy monetą, wówczas $h(\text{orzeł}) = h(\text{reszka}) = -\log \frac{1}{2}$, a zatem
$$H(X) = -\left(\frac{1}{2}\log\frac{1}{2} + \frac{1}{2}\log\frac{1}{2}\right) = \frac{1}{2} + \frac{1}{2} = 1 \text{ bit}.$$
*

2. Jeśli $X = \{x_1\}$, to $p_1 = 1$ i wówczas $H(X) = -1 \log 1 = 0$.

   Jeżeli ktoś wysyła nam komunikaty w kopercie, ale wiemy, że możliwy jest tylko jeden komunikat, to otrzymawszy kopertę, nie musimy jej nawet otwierać — wiadomo, co jest w środku. Zatem liczba bitów potrzebna do zapisania tego komunikatu jest równa 0. ◁

Uwaga — określenie *bit* ma dwa znaczenia:
- Po pierwsze, to jednostka ilości informacji, która oczywiście może być ułamkowa.
- Po drugie, to pojedynczy symbol z dwuelementowego alfabetu, niosący (co najwyżej) jeden bit informacji w pierwszym znaczeniu.

Ów dwuelementowy alfabet to tradycyjnie $\{0, 1\}$, ale rola tych symboli jest oczywiście tylko symboliczna. Na poziomie fizycznych urządzeń symbolom tym odpowiadają dwie różne wartości napięcia lub jakiejś innej wielkości fizycznej.

Entropia wyznacza zatem najmniejszą liczbę bitów (w drugim znaczeniu), która wystarcza do zapisania dowolnego komunikatu z rozważanego źródła.

Zdecydowanie najczęściej stosowaną w teorii informacji podstawą logarytmu jest $r = 2$, ale można również napotkać e i 10. Odpowiadające im jednostki noszą nazwy *nat* (ang. natural unit of information) i *dit* (ang. decimal digit).

Uwaga! Każda funkcja F, która:

1) jest ciągła jako funkcja prawdopodobieństw $p_i = P(x_i)$,

2) jest symetryczna: uporządkowanie komunikatów (tj. numeracja elementów zbioru $\{x_1, \ldots, x_n\}$) nie ma znaczenia, *

3) osiąga maksimum równe $\log n$, gdy wszystkie prawdopodobieństwa są równe,

4) jest addytywna: jeżeli podzielimy komunikaty na dwie niezależne części (tj. rozdzielimy komunikaty na krótsze ciągi symboli tak, że prawdopodobieństwa „lewych części" i „prawych części" są niezależne), to entropia całości równa jest sumie entropii części: *

$$F(XY) = F(X) + F(Y),$$

gdzie XY oznacza konkatenację komunikatów X i Y (kolejność nie ma znaczenia — „przychodzą dwie koperty"),

ma postać taką jak H zdefiniowana wcześniej (z dokładnością do stałego czynnika, czyli jednostki entropii wyznaczonej przez podstawę logarytmu).

Dowód powyższej własności można znaleźć w fundamentalnej pracy Shannona [7].

5.2 Kodowanie komunikatów i kompresja

Zastanówmy się teraz nad związkiem formy komunikatów z ich treścią, a ściślej — z ich prawdopodobieństwami. W praktyce chcielibyśmy przekazywać komunikaty w sposób oszczędny. Jeśli ustalimy odpowiedni kontekst porozumiewania się, to np. zamiast wysyłać komunikaty w formie zdań „w rzucie monetą wyrzucono orła" lub „w rzucie monetą wyrzucono reszkę", lepiej wysłać bit 0 lub 1. Innymi słowy, komunikaty odpowiednio kodujemy, przy czym oszczędne kodowanie oznacza taki dobór słów kodowych, by ich średnia długość była zbliżona do entropii.

Załóżmy zatem, że komunikaty kodujemy i przekazujemy za pomocą *słów kodowych* złożonych z bitów. Sytuacja jest więc następująca:

- Najprostszy pomysł na kodowanie to słowa kodowe równej długości. Jeśli możliwych komunikatów jest n, to potrzebujemy n takich słów. Żeby liczba kombinacji bitów była wystarczająca, słowa muszą więc mieć $k = \lceil \log n \rceil$ bitów.
- Zauważmy, że przy takim kodowaniu niektóre kombinacje k-bitowe mogą pozostać niewykorzystane. Będzie tak wtedy, gdy n nie jest potęgą dwójki.
- Zauważmy też, że takie kodowanie w ogóle nie zależy i nie wykorzystuje prawdopodobieństw poszczególnych komunikatów.
- Najwięcej informacji możemy przekazać, gdy komunikaty mają takie same prawdopodobieństwa.
- Jeśli tak nie jest (tj. prawdopodobieństwa komunikatów nie są takie same), to warto *przekodować* informacje z użyciem innych słów kodowych, oszczędzając na łącznej liczbie przekazywanych bitów.

Przykład 38. Załóżmy, że mamy do przekazania cztery możliwe komunikaty ze zbioru X, które kodujemy naiwnie jako 00, 01, 10 i 11. Ich prawdopodobieństwa wynoszą: $P(00) = \frac{1}{2}$, $P(01) = P(10) = \frac{1}{8}$, $P(11) = \frac{1}{4}$. Wówczas:

- Każde słowo kodowe ma długość równą 2 bitom.
- Entropia $H(X)$ wynosi:

$$\begin{aligned} -\left(\frac{1}{2}\log\frac{1}{2} + \frac{1}{8}\log\frac{1}{8} + \frac{1}{8}\log\frac{1}{8} + \frac{1}{4}\log\frac{1}{4}\right) &= \frac{1}{2} + \frac{3}{8} + \frac{3}{8} + \frac{2}{4} \\ &= 1\frac{3}{4} \\ &< 2 \text{ bity}. \end{aligned}$$

- Skoro entropia jest mniejsza niż długość słów kodowych, to da się zapisać tę samą informację oszczędniej, gdyż entropia jest najmniejszą średnią długością komunikatu (np. liczbą bitów), która wystarcza do przekazania komunikatów z danego źródła. *

- Komunikaty przekodujmy tak: *

$$\begin{aligned} 00 &\mapsto 0 \\ 01 &\mapsto 100 \\ 10 &\mapsto 101 \\ 11 &\mapsto 11 \end{aligned}$$

- Średnia długość komunikatu wyniesie teraz:

$$\frac{1}{2} \cdot 1 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 + \frac{1}{4} \cdot 2 = 1\frac{3}{4}.$$

- Osiągnęliśmy zatem minimum wyznaczone przez entropię, oszczędzając średnio ćwierć bitu. Lepiej już nie można — nie da się przekazać $1\frac{3}{4}$ bita za pomocą mniej niż $1\frac{3}{4}$ bita.

- Zauważmy jeszcze jedną własność nowych słów kodowych, bez której rozważane przekodowanie straciłoby sens: otóż żadne z tych słów nie jest prefiksem (przedrostkiem) innego słowa. Własność ta gwarantuje (choć nie jest konieczna), że odebrawszy ciąg „zlepionych" słów kodowych, możemy z niego jednoznacznie wydzielić poszczególne słowa. Nie są tu potrzebne dodatkowe separatory między słowami, które zniweczyłyby korzyść z przekodowania.

- Owa możliwość jednoznacznego wydzielenia słów z ciągu to podstawowa własność zbioru słów kodowych, bez której nie nadawałby się on do użytku. Oczywiście zbiór słów o jednakowej długości również ją (w trywialny sposób) posiada. ◁

Przykład 39. Przyjmijmy, że $P(x_1) = \frac{1}{4}$, $P(x_2) = \frac{3}{4}$. W takiej sytuacji entropia wynosi

$$-\left(\frac{1}{4} \cdot \log \frac{1}{4} + \frac{3}{4} \cdot \log \frac{3}{4}\right) \approx 0.811278 < 1.$$

Przy normalnym fizycznym sposobie przekazywania informacji nie można komunikatu przekazać przy pomocy mniej niż jednego bitu. Z drugiej strony, jeżeli komunikaty będziemy przesyłać wielokrotnie, to możemy znaleźć takie kodowanie, że długość słowa kodowego (w przeliczeniu na jeden komunikat) będzie dowolnie blisko entropii. W skończonym kodowaniu nie zawsze jest to możliwe. ◁

5.3 Średnia długość kodowania i redundancja

Definicja 5.3.1. Niech $X = \{x_1, \ldots, x_n\}$ będzie źródłem informacji, w którym komunikaty mają odpowiednio prawdopodobieństwa $p_1, \ldots, p_n$. Poszczególne komunikaty kodowane są ciągami znaków odpowiednio o długościach $l_1, \ldots, l_n$. Wówczas *średnią długością kodowania* nazywamy liczbę *

$$\ell = \sum_{i=1}^{n} p_i l_i,$$

zaś *redundancją kodowania* nazywamy różnicę między średnią długością kodowania a entropią źródła, czyli $\ell - H(X)$.

Warto pamiętać, że aby ta różnica miała sens liczbowy, jednostki w obydwu składnikach muszą być jednakowe. Zatem jeśli np. długości l_i (i w konsekwencji ℓ) są wyrażone w bitach, to entropia także powinna być w bitach.

Przykład 40. Mamy do przekazania cztery możliwe komunikaty ze zbioru $X = \{x_1, x_2, x_3, x_4\}$, których prawdopodobieństwa wynoszą $P(x_1) = \frac{1}{2}$, $P(x_2) = \frac{1}{4}$ i $P(x_3) = P(x_4) = \frac{1}{8}$. Entropia $H(X)$ wynosi zatem

$$-\left(\frac{1}{2}\log\frac{1}{2} + \frac{1}{4}\log\frac{1}{4} + \frac{1}{8}\log\frac{1}{8} + \frac{1}{8}\log\frac{1}{8}\right) \;=\; 1\frac{3}{4}.$$

Komunikaty kodujemy na trzy sposoby: *

1. $x_1 \mapsto 00, \quad x_2 \mapsto 01, \quad x_3 \mapsto 10, \quad x_4 \mapsto 11,$
2. $x_1 \mapsto 0, \quad x_2 \mapsto 100, \quad x_3 \mapsto 101, \quad x_4 \mapsto 11,$
3. $x_1 \mapsto 0, \quad x_2 \mapsto 10, \quad x_3 \mapsto 110, \quad x_4 \mapsto 111.$

Średnia długość słowa dla poszczególnych kodowań wynosi:

1. $\frac{1}{2}\cdot 2 + \frac{1}{4}\cdot 2 + \frac{1}{8}\cdot 2 + \frac{1}{8}\cdot 2 = 2,$
2. $\frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 3 + \frac{1}{8}\cdot 3 + \frac{1}{8}\cdot 2 = 1\frac{7}{8},$
3. $\frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \frac{1}{8}\cdot 3 + \frac{1}{8}\cdot 3 = 1\frac{3}{4}.$

Zatem redundancja poszczególnych kodowań wynosi:

1. $2 - 1\frac{3}{4} = \frac{1}{4},$
2. $1\frac{7}{8} - 1\frac{3}{4} = \frac{1}{8},$
3. $1\frac{3}{4} - 1\frac{3}{4} = 0.$ ◁

Narzuca się pytanie: jak otrzymywać kodowania o najmniejszej możliwej redundancji?

5.4 Kodowanie optymalne — algorytm Huffmana

W tym kodowaniu, komunikatom o większym prawdopodobieństwie przypisujemy krótsze słowa kodowe, zaś komunikatom o mniejszym prawdopodobieństwie — dłuższe.

Niech pary (x_i, p_i) oznaczają komunikaty i ich prawdopodobieństwa, przy czym $\sum_i p_i = 1$, zaś indeks i przebiega cały zbiór komunikatów. Poniżej opisujemy znany algorytm Huffmana, znajdujący kodowanie o możliwie najmniejszej długości.

Niech $S = \{(x_i, p_i)\}_i$.

Dopóki $|S| > 1$, wykonuj następujące czynności:

Znajdź w S dwie pary (a, p_a) i (b, p_b) o najmniejszym prawdopodobieństwie i zastąp je parą $(t, p_a + p_b)$, gdzie t jest drzewem złożonym z korzenia i dwóch liści a oraz b.

Uwagi do algorytmu:

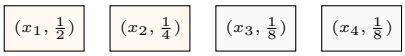
- Algorytm zatrzymuje się, gdy w zbiorze S pozostanie jedna para postaci $(t, 1)$.
- Drzewo t opisuje wówczas kodowanie komunikatów: ścieżka od korzenia do komunikatu wyznacza słowo kodowe, przy czym przejście do lewego poddrzewa to 0, a przejście do prawego poddrzewa to 1. *
- Algorytm wyznacza najlepsze możliwe kodowanie binarne: średnia długość słowa jest bliska entropii (lepiej nie można, gdyż nie mamy ułamkowych bitów).

Przykład 41. Załóżmy, że źródło wysyła cztery możliwe komunikaty o prawdopodobieństwach $\frac{1}{2}, \frac{1}{4}, \frac{1}{4}$ i $\frac{1}{8}$, czyli

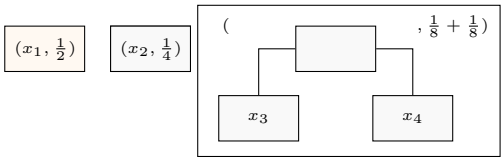
$$S = \{(x_1, \tfrac{1}{2}), (x_2, \tfrac{1}{4}), (x_3, \tfrac{1}{8}), (x_4, \tfrac{1}{8})\}.$$

Zawartość zbioru S w kolejnych krokach przedstawia się następująco:

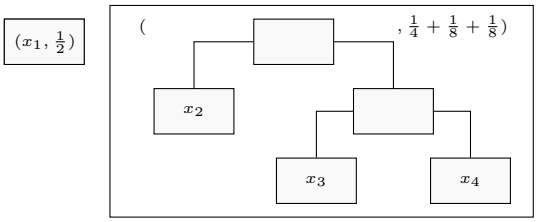
1. krok



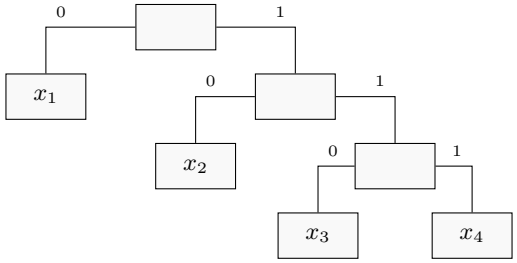
2. krok



3. krok



W ostatnim, czwartym kroku otrzymujemy drzewo:



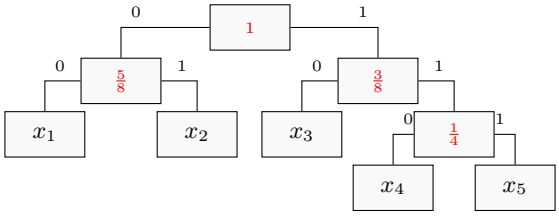
i odczytujemy z niego kodowanie optymalne:

$$x_1 \mapsto 0, \qquad x_2 \mapsto 10, \qquad x_3 \mapsto 110, \qquad x_4 \mapsto 111.$$

◁

W oczywisty sposób algorytm Huffmana nie jest jednoznaczny, tzn. może dawać różne kodowania dla tego samego źródła.

Przykład 42. Dla $S = \left\{\left(x_1, \frac{1}{4}\right), \left(x_2, \frac{3}{8}\right), \left(x_3, \frac{1}{8}\right), \left(x_4, \frac{1}{8}\right), \left(x_5, \frac{1}{8}\right)\right\}$ możemy otrzymać



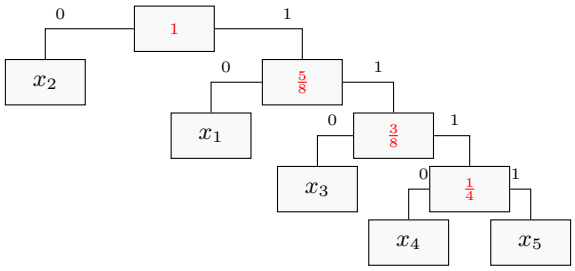
co daje kodowanie

$$x_1 \mapsto 00, \qquad x_2 \mapsto 01, \qquad x_3 \mapsto 10, \qquad x_4 \mapsto 110, \qquad x_5 \mapsto 111.$$

Średnia długość tego kodowania wynosi

$$\frac{1}{4} \cdot 2 + \frac{3}{8} \cdot 2 + \frac{1}{8} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = 2\frac{1}{4}.$$

Ale możemy również wziąć:



co daje kodowanie

$$x_1 \mapsto 10, \qquad x_2 \mapsto 0, \qquad x_3 \mapsto 110, \qquad x_4 \mapsto 1110, \qquad x_5 \mapsto 1111.$$

Średnia długość drugiego kodowania wynosi ponownie

$$\frac{1}{4} \cdot 2 + \frac{3}{8} \cdot 1 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 4 + \frac{1}{8} \cdot 4 = 2\frac{1}{4}.$$

◁

Bibliografia

[1] R.E. Bryant, D.R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, Pearson, 2015.

[2] E. Cygan, M. Ulas, *Algebra*, `im.uj.edu.pl/studia/md/doskonaly-uniwersytet`, Uniwersytet Jagielloński, 2021.

[3] J.E. Hopcroft, J.D. Ullman, *Wprowadzenie do teorii automatów, języków i obliczeń*, Wydawnictwo Naukowe PWN, 2003.

[4] P. Odifreddi, *Classical Recursion Theory*, vol. I (Studies in Logic, vol. 125), Elsevier, 1989.

[5] C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley Publishing Company, 1995.

[6] R. Penrose, *Nowy umysł cesarza*, Wydawnictwo Naukowe PWN, 1995.

[7] C.E. Shannon, A Mathematical Theory of Communication, *The Bell System Technical Journal*, vol. 27, 1948.

[8] M. Sipser, *Wprowadzenie do teorii obliczeń*, Wydawnictwo Naukowe PWN, 2020.

Skorowidz

A
alfabet, 11
    wejściowy, 15, 18
algorytm
    Euklidesa, 3
    Huffmana, 75
    sortowania
        przez prostą zamianę, 40
        przez proste wstawianie, 38
        przez proste wybieranie, 39
algorytmika, 3
automat, 25
    deterministyczny, 15
    liniowo ograniczony, 26
    niedeterministyczny, 18, 20
    ze stosem, 25

B
bit, 44, 72
    najbardziej znaczący, 44
    najmniej znaczący, 44
    znaku, 45, 50

C
cecha, 62
cykl, 12

D
długość (szerokość) słowa, 44
długość słowa, 11
dodawanie
    bez znaku, 53
    ze znakiem, 54
    zmiennoprzecinkowe, 68
drzewo, 13

E
ε, słowo puste, 11
entropia (źródła informacji), 72

F
funkcja przejścia, 8, 15, 18

G
graf
    nieskierowany, 12
    skierowany, 12
gramatyka, 25
    bezkontekstowa, 24
    kontekstowa, 25
    monotoniczna, 25
    nieograniczona, 26
    typu 0, 26
gwiazdka Kleene'ego, 17

H
hierarchia Chomsky'ego, 31, 32
hipoteza Churcha–Turinga, 9

I
IEEE 754, 62
ilość informacji, 72
inkrementacja, 52

J
język
    bezkontekstowy, 25, 31, 32
    formalny, 11
    generowany przez gramatykę, 24
    kontekstowy, 26, 31, 32
    regularny, 16, 20, 22, 31, 32
    rekurencyjnie przeliczalny, 26, 31, 32
    rekurencyjny, 26

K
klasa złożoności
    czasowej, 36
    pamięciowej, 36
kodowanie
    redundancja, 74
    słowo (kodowe), 73
    średnia długość (słowa), 74
    UTF-8, 46
komunikat, 71
konkatenacja, 11, 17
kontekst, 25
koszt algorytmu, 37

L
lemat o pompowaniu, 23

liść, 13

M
mantysa, 61
maszyna Turinga
  deterministyczna, 7
  niedeterministyczna, 35
mnożenie
  bez znaku, 58
  ze znakiem, 58
  zmiennoprzecinkowe, 67
model obliczeń, 9, 35

N
nadmiar, 55, 68
niedomiar, 68
  stopniowy, 65
nieliczba (Not-a-Number, NaN), 65
nieskończoność, 65

O
$\mathcal{O}$ duże, 33
o małe, 34

P
prefiks, 11
problem
  decyzyjny, 27, 30
  stopu, 28, 29
przesunięcie bitowe, 58

R
reprezentacja
  bez znaku, 44
  stałopozycyjna (stałoprzecinkowa), 61
  uzupełnieniowa (do dwóch), 44
  zmiennopozycyjna, 61
    podwójnej precyzji, 62
    pojedynczej precyzji, 62
rzutowanie (w języku C), 48, 49

S
słowo, 11
  puste ε, 11
stała (w języku C), 49
stan początkowy, 8, 15, 18
stopień niedeterminizmu, 35
sufiks, 11
symbol startowy, 24

Ś
ścieżka w grafie
  długość, 12
  nieskierowanym, 12
  skierowanym, 13

T
typ
  char, 46
  double, 66, 70
  float, 43, 66, 70
  int, 43, 44, 46, 70
  unsigned (int), 44

W
węzeł w drzewie, 13
wyprowadzenie w gramatyce, 24
wyrażenie regularne, 22

Z
zbiór
  krawędzi, 12
  L^*, 17
  nieterminali, 24
  produkcji, 24–26
  słów, 11
  stanów, 7, 15, 18
    końcowych, 8, 15, 18
  symboli, 11
    taśmowych, 7
    wejściowych, 7
  terminali, 24
  wierzchołków, 12
  wszystkich słów nad alfabetem, 12
złożoność
  algorytmu, 37
  czasowa, 33
  pamięciowa, 33
  problemu, 37
  sortowania, 39–42
znacznik (procesora), 53, 54

Ź
źródło informacji, 71